Scheduling Mechanisms for Efficient and Safe Automotive Systems Integration

Matthias Beckert

Dissertation
Technische Universität Braunschweig

Scheduling Mechanisms for Efficient and Safe Automotive Systems Integration

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines Doktors

der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von: Dipl.-Ing. Matthias Beckert

aus: Hildesheim

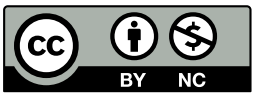
Eingereicht am: 30.04.2019

Mündliche Prüfung am: 25.10.2019

1. Referent: Prof. Dr.-Ing. Rolf Ernst
2. Referent: Prof. Dr.-Ing. Stefan Kowalewski

Druckjahr: 2019

Dissertation an der Technischen Universität Braunschweig
Fakultät für Elektrotechnik, Informationstechnik, Physik



https://doi.org/10.24355/dbbs.084-201911070747-5

Acknowledgements

The content of this dissertation was under development for almost seven years. It is quite obvious that you never walk such a long way alone. During the past years several companions supported me on my way to completion of this dissertation. Without the support of Prof Dr. -Ing. Rolf Ernst nothing of this would have been possible, since he gave me the opportunity to take this final step in my academical education.

The IDA is a well functioning institute and during my almost seven years as an employee I've learned very quickly how important the administrative part is. Without the help of Bettina Boettger, Sabine Klöpper, Gudrun Leuer or Peter Rüffer one would most likely drown in the bureaucracy of german public service. Being a researcher is nothing without fellows which share the same demanding way. When I started at the IDA in mid 2012, I wasn't even sure if I would make it to the finish line. Especially in the first months or even years, finding the right way of writing academic papers wasn't easy for me. Without the help of Moritz Neukirchner I'm not even sure if I would have kept on going. Under his supervision I was able to publish my first academic paper, kick-starting my "academic journey". And speaking of journeys, publishing at international conferences definitely has its benefits. I will always remember my first trips to the US with Philip Axer and Daniel Thiele. Same applies to the beer tasting with Johannes Schlatow in the crowded bars and streets of Seoul. The IDA in general is a very valuable source of knowledge because of it's very talented employees.

The daily discussions during lunch or coffee are something I miss, since I have left the institute. People like Holger Dinse or Mischa Möstl can provide you an answer to life, the universe and everything. And in terms of discussions, sharing interests in east european Sci-Fi literature with my long term office mate Adam Kostrzewa made even the most boring day interesting. During my time at the institute I supervised several students and one of them was more persistent than the others. Kai Gemlau started to work for me in the beginnings of his bachelor studies and later joined the institute as an employee after his master thesis. It was a pleasure to watch his development over the years, and I am grateful for the input and feedback he gave me for my work. Generally speaking, I am grateful to all my colleagues at IDA that I was able to spend so much time with such nice people. I don't regret a single minute of it.

One of the most common tasks a researcher has to deal with is, to create reviews for conference submissions. Well, what can I say? I can imagine more beautiful tasks. Creating reviews is a very time-consuming process and takes a lot of effort, especially if you're not familiar with the author's other works. Because of this, I am grateful that Prof. Dr. -Ing. Stefan Kowalewski invested the time to review this dissertation. One thing I often missed in the submissions I got for review were real-world test-cases from the industry. Sometimes this wasn't needed for the conference, but more often there were hardly any test-cases available at all. At this point I am happy that I had the opportunity to work with people like Hermann von Hasseln or Tobias Sehnke who both provided me a deep view into the design of existing automotive systems.

At the end of the day you go home after work, sometimes in good mood, sometimes in a bad mood. But always with the knowledge that your friends and family support what you are doing. I am grateful, that my parents Elisabeth and Frank, as well as my sisters Julia and Stefanie have always supported me in what have I done. I thank my friends Florian and Johannes for our regular boardgame and brewing sessions, which always provide the perfect counterpart to a stressful week. Same applies to the "Doko guys" Andy, Jonas, Carsten and Christoph for the shared time during our legendary weekend excursions. But most of all, I would like to express my deepest respect to my wonderful fiancée Janina. Together with our son Mika she always had my back and motivated me to finish my work. Thank you for supporting and enduring me the past seven years, even in my worst days.

Kurzfassung

Auf Grund moderner Themen wie dem assistierten Fahren steigt der Bedarf an Rechenleistung in aktuellen Fahrzeugen seit einigen Jahren kontinuierlich an. Um die erforderliche Rechenleistung zur Verfügung zu stellen, konzentrierten sich die Prozessorhersteller in der Vergangenheit auf den Wechsel von Einzel- zu Mehrkern-CPU Architekturen. Durch die zusätzliche Leistung ermöglichen solche Mehrkern-CPUs die Integration von zuvor einzeln ausgeführten Softwarekomponenten auf derselben Hardware, was zu einer Reduktion der Gesamtzahl an ECUs in Fahrzeugen beiträgt. Während Mehrkern-CPUs in Standard Computersystemen bereits seit langem Verwendung finden, ist der effiziente Einsatz in eingebetteten Systemen mit Echtzeitanforderungen häufig schwierig. Für die Integration von mehreren Softwarekomponenten auf derselben Hardware muss die Störungsfreiheit zwischen den einzelnen Komponenten für eine sichere Ausführung gewährleistet sein. Ein weiteres Problem besteht häufig bei bereits existierender Legacy-Software, welche für die Ausführung auf einer einzelnen CPU während der Entwicklung optimiert wurde und daher nicht ohne weiteres auf mehrere Prozessorkerne verteilt werden kann.

Diese Dissertation beschreibt zwei Mechanismen, welche eine sinnvolle Nutzung der zusätzlichen Rechenleistung ermöglichen sollen. Der erste Mechanismus verwendet partitionsbasierter Virtualisierung, welche in der Avionik bereits in der Vergangenheit in Form von ARINC653 Verwendung gefunden hat. Ziel ist hierbei, mehrere Softwarekomponenten auf derselben ECU zu integrieren.

Die Störungsfreiheit wird durch die Ausführung über einen Hypervisor erreicht, welcher die partitionsbasierte Virtualisierung implementiert und überwacht.

Zweitens wird die Integration des LET Paradigmas in eine automobile Systemarchitektur gezeigt, welches eine blockierungsfeie Synchronisation der Kommunikation über Kerngrenzen hinweg ermöglicht. Generell erlaubt dieser Mechanismus eine Synchronisation von Software über mehrere Prozessorkerne hinweg, was sowohl für die parallele Ausführung von Legacy-Software als auch von virtualisierten Partitionen genutzt werden kann.

Der wissenschaftliche Beitrag dieser Dissertation ist in erster Linie die Integration beider Mechanismen in einen automobilen Kontext. Dazu gehören eine Analyse der Antwortzeiten sowie eine Diskussion über bestimmte Herausforderungen bei der Implementierung. Beide Mechanismen werden in einer Prototyp Implementierung evaluiert und die Ergebnisse präsentiert.

Abstract

During the last years, the demand of computing power in modern cars has risen continuously, especially due to modern topics like assisted driving. In order to provide the required computing power, the chip manufactures focused in the past on the switch from single- to multicore CPU architectures. Such powerful multicore CPUs allow the integration of multiple software components on the same hardware, therefore reducing the overall number of ECUs inside a car. While multicore CPUs are well-known in general purpose computing, the efficient use in highly embedded systems with real-time requirements is more challenging. For the integration of multiple software components on the same hardware, freedom interference between those components must be enforced to ensure a safe execution. In case of existing legacy software the problem is often based on previously optimized execution for a singlecore CPU and as a result the parallel execution of such legacy software on multiple cores is not straight forward.

This dissertation describes two possible scheduling techniques in order to enable sensible use of the new available computing power. The first mechanism uses partition based virtualization, which has been a well-known technique in avionics with ARINC653. Objective is the integration of multiple software components on the same ECU. Freedom from interference is achieved through the execution on top of a hypervisor, implementing and monitoring the partition based virtualization. Second, an integration of the LET paradigm into an automotive system architecture, enabling a lock-less synchronized communication

across core boundaries. In general, such a mechanism allows a synchronization of software among multiple CPU cores. This allows synchronization across core boundaries which can be used for both, existing legacy software as well as virtualized partitions.

The scientific contribution of this dissertation is primarily the integration of both mechanisms into an automotive context. This includes a response time analysis as well as a discussion of certain implementation challenges. Both mechanisms will be evaluated in a prototype implementation, including a discussion of the results.

Contents

"DON'T PANIC!"

- The Hitchhiker's Guide to the Galaxy

CHAPTER 1
Introduction

The role of software and electronic hardware is becoming increasingly important in modern vehicles. Over the years the so called Electric/Electronic Architecture (EEA) of cars has evolved continuously. While the EEA itself is of no interest to the driver of a car, the functionality it implements defines the driving experience. The EEA consists of a set of Electronic Control Units (ECUs) which are connected via a set of buses and/or networks. Each ECU houses software which either provides some specific functionality or is part of a bigger function which is distributed on several ECUs.

Current and presumably future ECUs are developed based on the International Standard 26262: "Road vehicles - Functional safety" (ISO26262) [66], which represents the standard for functional safety of road vehicles. The ISO26262 was developed based on the generic International Electrotechnical Commission, Standard 61508: "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems" (IEC61508) [65] taking into account the requirements and needs of the automotive industry. Comparable standards to the ISO26262 are also available in other domains. As an example, in avionics the DO-178B, Software Considerations in Airborne Systems and Equipment Certification (DO-178B) [94] and its counterpart DO-254, Design Assurance Guidance for Airborne Electronic Hardware (DO-254) [95] define functional safety for software and hard-

	ASIL			
	A	B	C	D
SPFM	-	$> 90\%$	$> 97\%$	$> 99\%$
LFM	-	$> 60\%$	$> 80\%$	$> 90\%$
PMHF	-	$< 10^{-7}h^{-1}$	$< 10^{-7}h^{-1}$	$< 10^{-8}h^{-1}$
FIT	-	< 100	< 100	< 10

Table 1.1: ASIL hardware fault metric specification

ware components in airborne systems. ISO26262 defines functional safety as "absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems" [66].

One of the major definitions of the ISO26262 are Automotive Safety Integrity Levels (ASIL), which are the counterpart to the Safety Integrity Levels (SIL) from IEC61508. The ASIL classification is used to define automotive-specific hardware requirements in order to preserve functional safety. There are four integrity levels available, starting with low safety requirements in ASIL A up to the most stringent safety requirements in ASIL D. In general, ISO26262 doesn't assume a system to be perfect without any kind of hardware or software faults. Instead, the incidence of allowed faults is limited. Table 1.1 gives a fault metric overview of the different integrity levels based on random hardware failures.

The first entry in Table 1.1 denotes the Single-Point Fault Metric (SPFM). SPFM indicates the robustness against untrapped hardware faults that would cause an immediate violation of functional safety. The second entry denotes the Latent Fault Metric (LFM). Again, LFM indicates the robustness against untrapped hardware faults. But instead of immediate, latent violations are taken into account. As an example for ASIL C, a maximum of 3% of all untrapped single-point and 20% of all untrapped latent faults are allowed to cause violations of functional safety. Next the Probabilistic Metric for random Hardware Failures (PMHF) describes the overall number of allowed untrapped hardware failures per hour. For a better representation this can be converted into Faults In Time (FIT), which denotes the number of allowed failures per one billion hours.

Another important requirement from ISO26262 is the so called freedom from interference defined as "absence of cascading failures between two or more elements that could lead to the violation of a safety requirement" [66]. In case of software components this means, that a failure in component A will never cause a failure in component B and vice versa. Although the ISO26262 does not define any specific mechanisms to it, it makes sense to take a look at possible sources of danger with regard to freedom from interference. In general there are three different sub-areas that need to be considered.

Temporal Isolation: The temporal behavior of component A is independent of component B, even if component B crashes while blocking the Central Processing Unit (CPU) or actively tries to disturb component A. Usually an operating system ensures temporal isolation.

Spatial Isolation: A memory access of component B can never overwrite private data of component A, unless otherwise specified. This must be also the case, even if component B access the memory accidentally through a wild pointer. Usually spatial isolation is ensured by the CPU in hardware with a Memory Protection Unit (MPU) or Memory Management Unit (MMU).

Deadlock prevention: The access to shared resources or peripherals is usually coordinated with mechanisms like semaphors. In order to provide deadlock prevention, it must be ensured that a crashed component cannot block a shared resource permanently. A known technique for deadlock prevention is the use of watchdogs [15].

All of these sub-areas rely on a combination of hardware and software mechanisms. While classical temporal isolation usually requires only a timer with a sufficient resolution as time-base, the corresponding software executed during runtime is often much more complex. On the other hand, spatial isolation is primarily ensured in hardware. The corresponding software is often limited to an accurate configuration of MPU or MMU during startup. For watchdogs different implementations from tick based software implementations to distinct hardware modules are possible. In general, achieving freedom from interference always requires a combination of special hardware and software mechanisms.

As the number of functions in a car increases, also the load on each ECU and interconnect rise. To implement further more functionality in a car, the used hardware inside an ECU must be replaced in order to gain processing power. Since a few years, the evolution of processors in embedded systems has gone the same way as in consumer electronics. The processors clock speed as well as the number of integrated cores has increased. As a result, ensuring freedom from interference gets more and more important, since otherwise single faults may have an even bigger impact to the system. Nevertheless, faster processors do not reduce the loads on the interconnects. The migration from single to multicore CPUs is also challenging. Often existing legacy software developed for singlecore CPUs is reused, which might lead to problems in case of coordination and data consistency.

With respect to temporal and spatial isolation, integrating further more functions on a single ECU gets even more sophisticated. As already mentioned, spatial isolation is usually achieved through the use of hardware functionality like memory protection or management. When implementing spatial iso-

lation this way, it is required to reconfigure the corresponding hardware module (MPU/MMU) each time a different software component is executed. In the worst-case this means that the hardware must be reconfigured on each switch between the different software components, which can introduce major runtime overhead.

A counter measure to this additional overhead can directly be derived from the fact that most of the executed code is existing legacy software. Usually, existing legacy software has been executed flawlessly before on a single ECU, often without any kind of hardware based memory protection. Instead, freedom from interference has been ensured through extensive testing and verification. With a higher number of software components on a single ECU, the costs and expenses for testing and verification rise. A possible method to mitigate this could be partitioning. General idea is to bundle multiple software components into larger partitions. The verification is then performed separately on each single partition, with a much smaller number of software components inside. Ideally this verification has been already performed before for an older ECU but the same combination of software components. The handling of different partitions is then performed by a software which provides freedom from interference between partitions. A possible way to provide temporal as well as spatial isolation between different partitions is to use an embedded hypervisor for virtualization.

Virtualization is a well-known approach in the field of general purpose computing. Often one physical machine hosts 20 virtual ones or even more. Since hardware in the embedded domain gets more and more powerful, techniques like virtualization gain attention. Especially in domains which are not as cost driven as the automotive sector, virtualization with more powerful CPUs was already considered some time ago. With the Integrated Modular Avionics (IMA) architecture and Aeronautical Radio Incorporated Avionics Application Standard Software Interface (ARINC653) [93] as a corresponding implementation description, virtualization techniques have become part of a standardized software architecture in safety-critical systems. This was possible, as in avionics the costs per unit aren't as important as in the automotive industry due to much higher system costs compared to the electronic and software parts. However, combining multiple computation units on a single hardware reduces the overall weight, which is much more important for an airplane.

Figure 1.1 shows the basic setup of two virtualized applications according to ARINC653. Each application is encapsulated in a partition, executed with minimized privileges in the user mode of the CPU. In addition to an application, each partition might also contain an Partition Operating System (OS) (POS). The privileged system mode of the CPU is reserved for the hypervisor which provides access to the underlying hardware. With the Application Executive (APEX) [107] Application Programming Interface (API), an ARINC653 conformant hypervisor

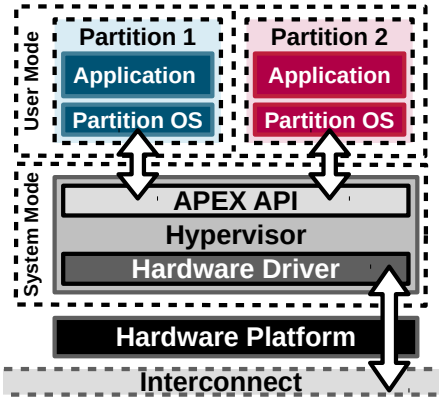


Figure 1.1: Simplified hypervisor based system setup

provides a set of functions to access hypervisor functionality from a partition[93]. Commercial hypervisor implementations like PikeOS [70] or VxWorks 653 [116] usually provide an own POS or API implementation in addition to the APEX API. As an example, both mentioned implementations provide also a Portable OS Interface (POSIX) for virtualized applications. Also, PikeOS has already been considered as a possible hypervisor for the automotive domain [29], despite its origins in avionics.

As shown in Figure 1.1 applications are isolated from each other. The hypervisor is responsible at that point to ensure spatial and temporal isolation. As mentioned before, spatial isolation is usually achieved based on either an MPU or MMU. A side effect of the use of memory protection or management for spatial isolation is, that not only a safe but also secure execution of software inside a partition is achieved. Resulting from this, the use of MPU or MMU enables a secure execution of possible intellectual property of different software suppliers on the same ECU. With the hypervisor occupying the CPU's system mode, it has full control of either MPU or MMU. To provide spatial isolation, the hypervisor simply reconfigures the corresponding register when switching execution between different partitions based on a static configuration.

While spatial isolation is primarily based on core features of a modern CPU, temporal isolation instead depends primarily on the implemented scheduling algorithm. Often only an accurate timebase is needed to implement a scheduling strategy. Nowadays, even in the smallest Microcontroller (μC) such a timebase can be realized with a timer module. A simple way to describe temporal isolation is, that the time a software component needs to finish its computation, is independent of the behavior of all other software components on the same

ECU. We provide a more versatile and formal description of temporal isolation in Section 4.1.

ARINC653 achieves temporal isolation through a simple Time Division Multiple Access (TDMA) scheduling on partition-level. In general, the different partitions get executed in a static order for fixed amount of time. An obvious drawback of such a static execution is the handling of sporadic events like Interrupt Requests (IRQs). As a simple example based on Figure 1.1, if the ECU is executing partition 1, IRQs for partition 2 can not be processed during that time. This is even the case if partition 1 has no work to do. Chapter 4 explains this issue in much more detail. In case of avionics, this isn't as bad as expected. First of all the entire IMA is a uniform architecture with interconnects like ARINC429/629 or Avionics Full DupleX Switched Ethernet (AFDX) between different ECUs, which match the partition-level TDMA scheduling. Second, usually avionics aren't that timing sensitive and as a result, sensors or actors might be processed based on polling and not on IRQs due to better predictability. And third, if a sensor or actor needs a close interaction to an μC in order to work properly, it is encapsulated with an own μC and connected to the already mentioned interconnect, which again matches the partition-level TDMA scheduling.

Adding additional hardware is possible due to the less cost sensitivity in avionics. This is different for the automotive domain, which has a much higher volume and costumers which are usually not willing to spend a lot of money. As a result, the automotive industry tries to avoid unnecessary ECUs at that point and often connects sensors or actors directly. Also, IRQs are much more timing critical in the automotive domain. As an example, if a crash of a car is registered, safety mechanisms like seatbelt tensioner or airbags must be activated within a few milliseconds. But not only in a worst-case scenario like a crash, also in normal driving situations a fast processing of IRQs is absolutely necessary. Let's assume the driver wants to accelerate the car and pushes the pedal to the metal. The behavior of the car expected by the driver is to accelerate within a fraction of a second. What the driver doesn't know is, that such a simple command might involve several ECUs inside a car for input processing, torque coordination and engine control. And inside all of those ECUs the processing must be fast in order to enable the expected response behavior of the car. Without a fast IRQs processing, such a behavior is hard to achieve.

Beside the need of a fast IRQ processing, there is also another problem. The parallel execution of software on multicore CPUs isn't addressed by ARINC653, since the definition only include singlecore CPUs. In order to deploy an automotive application on a multicore platform, it is necessary to know the dataflow between different software components. The best case would result in a system, where the software on different cores is completely independent of each other. Practically this is often out of reach and dependencies exists between the tasks

on different cores. Thus, core-to-core communication should be minimized based on an extensive dataflow analysis of the system software, but assuming that it can be eliminated completely is way too optimistic.

A possible method, how dependencies can be identified in automotive legacy software, has been introduced in [59]. The proposed algorithm groups independent software parts, which can run in parallel without additional synchronization. Dependent software parts are then sequenced after each other. For each dependency across multiple cores a synchronization method or special communication mechanism is needed. Often hardware architectures in the automotive industry do not support special core-to-core communication mechanisms. For minimal overhead, lock-free communication based on shared memory variables is often used. Such a lock-free communication can be implemented in two different ways. The simple solution is a "don't care" behavior for input variables. In this case it does not matter if a read-after-write behavior is enforced or not. But such an approach only works for specific algorithms. A possible solution to this is the Logical Execution Time (LET) paradigm, which defines global points in time when data is written or read. This way, only a global time is needed to synchronize the software on different cores. Even though the LET paradigm is already considered to be part of the Automotive Open System Architecture (AUTOSAR) in the future, an open question is still the versatility of LET and the way how it can be integrated into the existing architecture.

1.1 Contribution

In general, utilizing the newly available computing power is challenging in different ways. On the one hand, sharing a powerful ECU with multiple applications may violate freedom from interference. On the other hand, distributing a previous singlecore applications to multiple cores may not be that easy either. The contribution of this work is therefore structured in two main topics.

First, we propose a system architecture which provides virtualization mechanisms with real-time capabilities. The proposed system architecture is based on the ARINC653 standard, which relies on a static time partitioning. In order to achieve a better IRQ performance we provide different modifications to the original architecture. This includes a monitoring based IRQ shaping, an optimization algorithm for time partitions and a Sporadic Server (SPS) based budget scheduling on partition-level. As a formal proof, we cover all proposed mechanisms with an Response Time Analysis (RTA) based on Compositional Performance Analysis (CPA) implementing the well-known multiple event busy-window [77, 104]. The paper published in context with this work are [28, 23, 26, 24].

Second we take a look the LET paradigm as a possible coordination instance

in order to execute existing automotive software on multicore processors. The applicability of the LET paradigm is discussed and an integration method is proposed. This includes additional methods for an RTA as well as an upper interference bound. The work on this topic is primary based on [27], [25] and a talk given at [44].

For both topics we intensively discuss the implementation challenges, with focus on efficiency on existing hardware. As a result of the discussion we introduce working implementations for both topics, which are then used to evaluate the proposed mechanisms.

1.2 Outline

The remainder of this dissertation thesis is structured as followed. Chapter 2 gives an overview of the current architecture of modern ECUs. This includes the hardware as well as the software architecture. At the end of this chapter, we finalize the problem statement based on existing hardware and software architecture in the automotive domain. Chapter 3 introduces the system model. For this purpose, the general RTA framework and used notation is explained.

Chapter 4 provides an overview of the scheduling technique described by ARINC653. Also, possible problems regarding IRQ handling are explained, as well as a possible solution is introduced. In order to generate valid system configurations, we also provide a partition size optimization based on a RTA for ARINC653 based system. Chapter 5 shows how to map the system explained in Chapter 4 to an SPS based partition scheduling. In order to provide the same degree of sufficient temporal independence as for ARINC653, we introduce a budget based partition scheduling in combination with the SPS mechanism. The budget based scheduling is then extended to a fully work conserving scheduler. All provided mechanisms in this chapter are again covered with an RTA. Chapter 6 gives an introduction to the LET paradigm and shows how this can be applied to an automotive software architecture. This includes the discussion of a task distribution strategies and possible integration mechanisms.

Chapter 7 discusses the implementation challenges of the mechanisms described in Chapter 4, 5 and 6. Here we focus on specific aspects relevant for automotive systems. This includes primarily the described hardware/software architecture from Chapter 2 as well as general programming techniques for code with improved temporal predictability. Chapter 8 evaluates the proposed mechanisms. This includes the actual implementations as well as the provided RTA. Chapter 9 summarizes the outcome of the dissertation and outlines possible future work.

"Software is like sex: It's better when it's free"

- Linus Torvalds

CHAPTER 2

Trends in Automotive ECU Architectures

Main task of this chapter is to convey some basic knowledge about the architecture of current and future ECUs, in order to understand the later finalized problem statement and proposed solutions. The chapter starts with an inside into the hardware and software architecture of modern ECUs. This includes two multicore architectures which can be considered as examples for different automotive use-cases. On the software side, an insight into the structure of automotive applications and the surrounding Runtime Environment (RTE) is given. Based on both, the underlying hardware and the existing software framework, the already started problem statement from Chapter 1 is finalized.

2.1 Hardware Architecture

Figure 2.1 gives a short overview of the essential hardware elements of a modern ECU. The mechanical structure of an ECU is based on a Printed Circuit Board (PCB) inside a standardized housing with an ECU-specific connector. Usually the most important part of an ECU is the central μC or in some cases an System On a Chip (SoC). Both, μC and SoC describe systems including one or more CPUs, integrated peripherals and memory controllers. A μC usually directly includes *Flash* and *RAM*, while those are often separate for some SoCs. Therefore, external

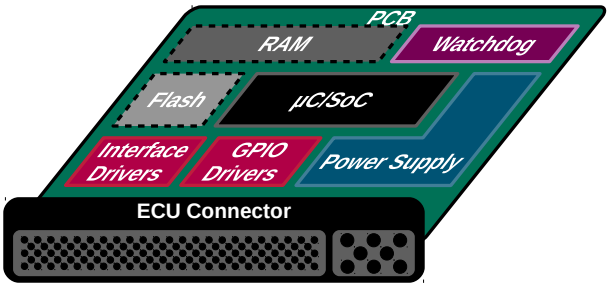


Figure 2.1: Basic structure of an ECU

Flash and *RAM* don't need to be necessarily a part of the ECU. A hard separation between μC and SoC isn't always straight forward these days. More often the term SoC is used by the manufactures for marketing purpose. For the sake of clarity, we also use the term SoC as a general description in this dissertation.

The *Power Supply* is also a major part of an ECU, and occupies a lot of space on the PCB. An external *Watchdog* is also often part of the ECU in order to implement special power and external behavior monitoring. The connectivity to the outside of the ECU housing is provided by special physical *Drivers*. *Interface Drivers* connect the physical layer of the used interconnect to the μC/SoC. This can range from a simple voltage-level conversion to an extensive timing behavior. The *GPIO Drivers* are often based on power transistors controlled by the μC/SoC in order to switch an external load.

As shown in Figure 2.1, the functional scope of an ECU is primarily defined by the used μC/SoC. For a better comparison, we use the following parameters to describe the different μC/SoC and ECU architectures. First, the CPU architecture, including basic structure, Instruction Set Architecture (ISA), multicore integration and internal interconnects. Second, the memory subsystem, with primary focus on Random Access Memory (RAM) connection and caching. Third, the available external interconnect interfaces and other use-case specific peripherals. As last parameter, we compare implemented features which support adherence of functional safety.

2.1.1 AURIX

The AURIX family is a μC architecture from Infineon [3], primarily developed and targeted for the automotive domain. Figure 2.2 shows the general structure of the AURIX family based on the TC275, which represents the upper mid-range of the first generation's AURIX family. Developed regarding ISO26262 require-

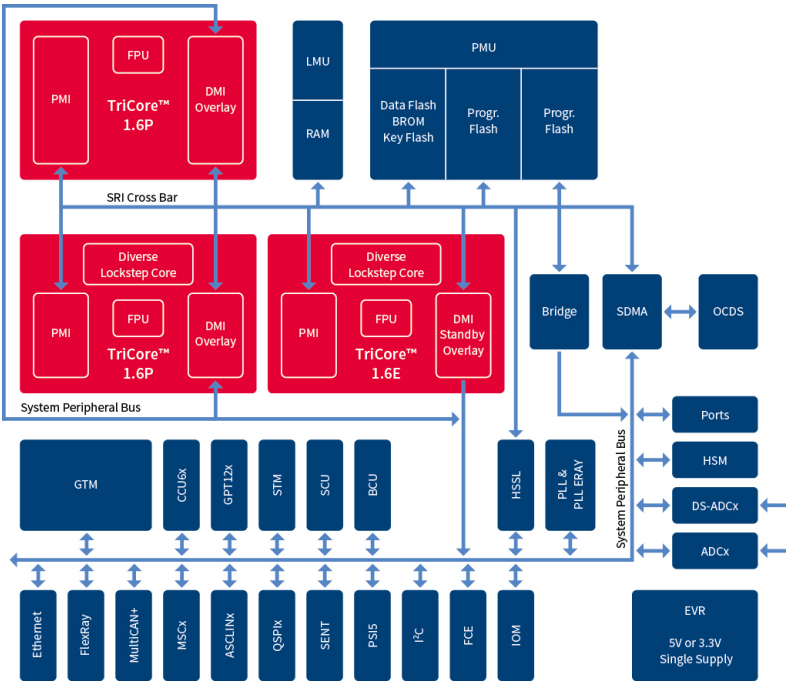


Figure 2.2: AURIX 1 generation μC architecture [4]

ments the AURIX family allows certification of up to ASIL-D [3]. Although first details about the second generation AURIX are already available [3], we primarily focus on the details of the first generation as those are available to the public. Therefore, all data mentioned refer to the first generation AURIX family unless otherwise stated.

CPU architecture and SoC structure

The AURIX family is a 32-bit Reduced Instruction Set Computer (RISC) architecture, implementing multiple TriCore [5] CPUs on the same chip. There are different hardware series available, implementing either two or three TriCore CPUs for the first and up to six CPUs for the second generation AURIX SoCs. The maximum clock frequency of the integrated TriCore CPUs is 300 MHz and depends on the actual series.

The example from Figure 2.2 shows a setup with three TriCore v1.6 CPUs. Each TriCore consists of a general purpose CPU, a Floating Point Unit (FPU)

and a Digital Signal Processor (DSP). For the shown TC275, two different variants of the TriCore CPU are used. An energy optimized v1.6e and a performance optimized v1.6p. The difference between both versions is primarily the pipeline structure. The actual combination of TriCore CPUs depend on the AURIX variant. As an example, the TC295 implements three v1.6p CPUs compared to the TC275 with one v1.6e and two v1.6p. The different TriCore CPUs are connected among each other with a System Resource Interconnect (SRI) cross bar, which also connects the CPU-external memory subsystem and a Direct Memory Access (DMA) controller. Additional periphery is available through the system peripheral bus.

Memory Subsystem

The memory subsystem can be divided into CPU-internal and external memory. As the actual sizes for both types differ for various AURIX variants, we state the maximum sizes. It is also important that the first generation AURIX μCs do not provide any interface for off-chip memory.

As mentioned before, the CPU-external memory is connected via the SRI. It consists of a Program Memory Unit (PMU) and a Local Memory Unit (LMU). The PMU provides program flash memory and a data flash memory for non-volatile variables. Overall size of the PMU ranges up to 8 MB program and 768 kB data flash. The LMU provides a global Static Random-Access Memory (SRAM) with 32 kB memory.

Beside PMU and LMU, each TriCore CPU also contains an internal SRAM and additional caches. The CPU-internal SRAM is divided into Program Scratch-Pad SRAM (PSPR) and Data Scratch-Pad SRAM (DSPR). Both, PSPR and DSPR are tightly coupled to the corresponding TriCore CPU and provide a fast memory access. As all memories share the same address space, remote access to PSPR or DSPR of another TriCore CPU through the SRI is also possible. The maximum SRAM sizes for PSPR/DSPR are 32 kB / 240 kB per TriCore CPU. Each TriCore CPU also provides its own data and instruction cache, implemented as SRAMs. While an access to the CPU-internal SRAM does not achieve any speed up through the cache, it is noticeable for remote accesses through the SRI. Important is at this point, that there is no coherency protocol implemented in the caches. As a result global data, shared across multiple TriCore CPUs, should not be cached. Otherwise, inconsistent data might be the outcome. The maximum cache sizes for data/instruction cache are 8 kB / 32 kB. As the caches are implemented as SRAMs, they can be deactivated and used as an extension to PSPR and/or DSPR.

Interconnects and peripherals

The AURIX family provides a wide range of interconnects from the automotive industry. The Local Interconnect Network (LIN) provides the lowest bandwidth and is represented as *ASCLIN* module in Figure 2.2. In case of the top of the range implementation the AURIX provides four LIN interfaces. Next there is the *MultiCAN+* module, implementing both Controller Area Network (CAN) and CAN with Flexible Data-Rate (CAN-FD) interfaces. The maximum accumulated number of either CAN or CAN-FD is limited to six interfaces implemented by two *MultiCAN+* modules. In the case of higher bandwidth, likely used as a backbone interconnect of different domain, the AURIX supports two *FlexRay* interfaces. For even more bandwidth, the AURIX also provides one 100 MBit/s *Ethernet* Media Access Controller (MAC).

The number of available interconnects directly defines one use-case for the AURIX family. This use-case is the integration as central gateway or central controller ECU with connection to several car domains [64]. Beside communication, also other peripherals define specific automotive use-cases. As an example, Capture Compare Unit 6 (CCU6) and Generic Timer Module (GTM) are two powerful timer modules, which can be used for various kind of signal generation. Especially the generation of Pulse Width Modulated (PWM) signals to drive brushed or brush-less Direct Current (DC) motors can be covered entirely by the CCU6 or GTM [64]. Due to the redundant CPU configuration, the AURIX family is also suitable for safety applications like steering, braking or airbags [64]. As Advanced Driver Assistance Systems (ADAS) get more and more important in future systems, the second generation AURIX also integrates a new radar sub-system for fast processing of environmental data.

Figure 2.2 shows more, than the already mentioned interfaces and peripherals. Nevertheless, we stop here and refer to Infineons AURIX documentation for more details [3]. Although for most of the AURIX variants the documentation is confidential, the TC275 user manual is freely available [4].

Safety Features

The AURIX family implements different mechanisms in order to achieve ASIL-D. One of these mechanisms is called *diverse lockstepping*. The variant shown in Figure 2.2 implements this feature on two of the shown TriCore CPUs. In case of lockstep execution, both TriCore CPUs execute the same instructions with a two cycle delay. A comparator logic checks if both CPUs provide the same results and therefore detects computation faults. Due to the two cycle delay, faults based on environmental influences (like bit flips) can be detected.

Another relevant safety feature we would like to mention with regard to spa-

tial isolation is the implemented memory protection. The AURIX uses a two level memory protection mechanism implemented on CPU and bus-level [47]. First, each TriCore CPU implements a region based MPU which provides spatial isolation between software components on the same CPU. As only a single address space is supported on the AURIX, each TriCore CPU can access PSPR and DSPR of another CPU. Such remote access is not covered by the CPU internal MPU of each TriCore. Instead, a second bus-level MPU is implemented in each SRI interface. Both, CPU and bus-level MPU are configured through memory mapped registers. In order to prevent reconfiguration during run time, write access to those configuration registers can be blocked in hardware after an initial configuration during start up. Beside MPU configuration registers, also the access rights to all other peripheral registers can be controlled. This way the AURIX provides spatial isolation, although the entire memory and all configuration registers share the same single address space.

2.1.2 R-Car H3

The R-Car H3 is a SoC developed by Renesas Electronics, targeting the automotive domain. Developed regarding ISO26262, the R-Car H3 allows a certification up to ASIL-B [6]. The R-CAR H3 represents the third generation of Renesas automotive computing platforms and offers the highest computing power compared to other R-Car variants. Details given in this subsection are based on [6], [97] and [101]. Figure 2.3 shows the general structure of the R-Car H3 SoC.

CPU architecture and SoC structure

In contrast to the AURIX from Infineon, Renesas decided not to design an own CPU architecture for the R-Car H3. Instead, they use different CPU designs from ARM [1]. Beside other minor topics, ARMs main business is the design of CPUs including ISA and implementation.shown ARM itself doesn't own an in-house fabrication process for its proposed CPU designs. Instead, companies like Renesas purchase CPU licenses, integrate those into their peripheral ecosystem and fabricate the resulting SoC on a silicon die. An ARM CPU always implements a specific version of the ARM ISA. Since version 7, ARM specifies three different variants of its ISA targeting different scopes.

ARMv(7,8)-A: Implemented by the Cortex-A series of ARM CPUs, designed for application processors with maximized general purpose computing power

ARMv(7,8)-R: Implemented by the Cortex-R series of ARM CPUs, designed for real-time processors with optimized predictability

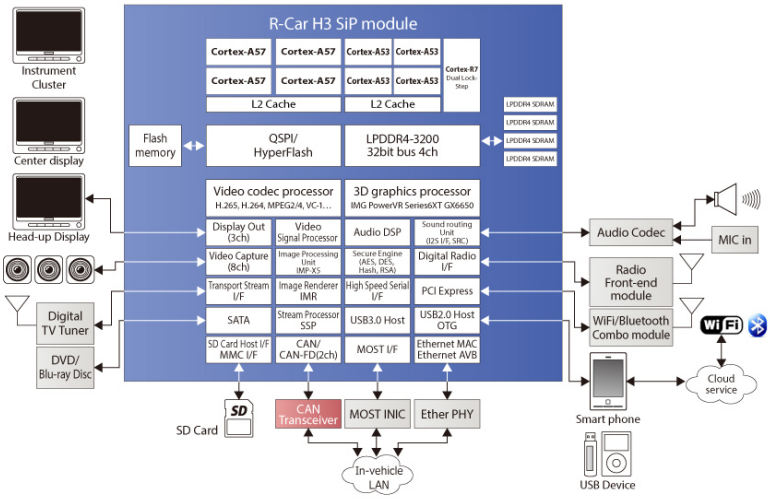


Figure 2.3: R-CAR H3 SoC architecture [6]

ARMv(7,8)-M: Implemented by the Cortex-M series of ARM CPUs, designed for μCs with low power consumption, optimized cost and low latency behavior in deeply integrated embedded systems.

In some cases companies develop their own CPU designs, which behave according to a specified version of the ARM ISA. This way the cost for licensing can be reduced.

The R-Car H3 integrated three different types of ARM CPUs [97]. A cluster of four Cortex-A57 CPUs, a second cluster with four Cortex-A53 CPUs and a third cluster with two Cortex-R7 CPUs. Both Cortex-A clusters implement the ARMv8-A ISA which represents a 64-bit RISC architecture. Such a CPU combination is proposed by ARM as big.LITTLE [9] concept, where a high-performance CPU (A57) is combined with a high-efficiency CPU (A53). Due to the identical ISA, software can be migrated during runtime between different cores in order to achieve either higher performance or higher energy efficiency. Cortex-A57 as well as Cortex-A53, both provide a MMU and hardware supported software virtualization. This is especially helpful if the virtualized software contains a more complex POS like a Linux or something POSIX compliant. Complex operating systems isolate the kernel inside the processor's system mode. Interaction with the kernel from the application is achieved usually through a system call interface. On a hypervisor this cannot be achieved directly, as the system mode is

reserved for the hypervisor. As a result the kernel of the POS is executed in the same context as the application without any isolation. Also, on a complex operating system multiple address spaces for different parts of the applications inside a partition are used. In order to change the address space inside a partition, the POS would need access to the MMU configuration registers which are under control of the hypervisor. A possible solution is to provide an additional function call to the hypervisor in order to change the address space from a partition layer, resulting in additional runtime overhead. The implemented hardware support for virtualization in the R-Car H3 (and other CPUs based on the same ISA [10]) solves both problems by providing an additional CPU mode with privileged hardware access rights and a multilayer memory translation with intermediate addresses inside the MMU. This way, the POS can execute in the processor's system mode and control the first level memory translation, while the hypervisor uses the even more privileged mode with access to the second level memory translation.

Each CPU is clocked with either 1,5 GHz (A57) or 1,2 GHz (A53). The third cluster is based on two Cortex-R7 CPUs, implementing a 32-bit RISC architecture according to the ARMv7-R ISA, clocked with 800 MHz. All three clusters are connected among each other, as well as with the available peripherals, via ARMs Advanced eXtensible Interface (AXI) as part of the Advanced Microcontroller Bus Architecture (AMBA). Cache coherency among all CPUs is ensured via the AXI Coherency Extension (ACE).

Memory Subsystem

The memory subsystem of the R-Car H3 provides several memory layers. First, each CPU uses caching in different configurations. Each Cortex-A57 CPU has two level 1 caches for instructions and data, providing 48 kB and 32 kB. Same is the case for each Cortex-A53 CPU with 32 kB of instruction and 32 kB data cache. Both Cortex-A clusters provide also an own level 2 cache. Size in case of the Cortex-A57 cluster is 2 MB and 512 kB in case of the Cortex-A53 cluster. The Cortex-R7 cluster is different at this point. Like both Cortex-A clusters, it provides level 1 instruction and data caches with 32 kB each per CPU.

The main RAM interface of the SoC supports up to 8 GB of external Low Power Double Data Rate Fourth-Generation Synchronous Dynamic Random-Access Memory (LPDDR4 SDRAM). In case of fixed internal memory, also 384 kB SRAM is available. For fast and predictable memory access the Cortex-R7 cluster implements Tightly Coupled Memory (TCM) which is comparable to the AURIXs scratch-pad RAMs. Each Cortex-R7 CPU implements 32 kB TCM for instructions and 32 kB TCM for data. In case of non-volatile memory, the SoC provides several interfaces for external flash memories. Internal flash memory is not available inside the SoC.

Interconnects and peripherals

On side of the classic automotive interconnectes, the R-Car H3 supports two CAN channels which also provide CAN-FD functionality. Additionally, one Media Oriented Systems Transport (MOST) interface is available. For higher bandwidth, the R-Car H3 supports 1GBit Ethernet with additionally Audio/Video Bridging (AVB) support.

A closer look on Figure 2.3 shows directly the application focus of the R-Car H3, which is video and audio processing. The SoC contains a dedicated 3D graphics processor running at 600 MHz and additional hardware video decoders. Additionally to video processing, also hardware for video capture and display output is available. In case of audio, a dedicated digital signal processor and a several output interfaces are available. A possible use-case for the R-Car H3 due to the rich multi media support is infotainment or head unit, implementing a car's Human Machine Interface (HMI).

Safety Features

The basic safety features of the R-Car are comparable to the AURIX. The Cortex-R7 cluster supports a lockstep mode and uses a fast TCM. Due to the local TCM, interference from any Cortex-A CPU during memory accesses is eliminated as long as the data is stored inside a TCM. In case of spatial isolation the different clusters support either a MMU (A57 & A53) or MPU (R7). Outside of a CPU the R-Car provides an additional bus-level MMU. The entire R-Car H3 allows a system certification of up to ASIL-B.

2.1.3 Comparison

Table 2.1 summarizes some key features of the AURIX and R-Car H3 architectures. While both presented hardware architectures implement similar mechanisms to support functional safety according to ISO26262, the use-cases inside the EEA strongly differ. The AURIX is designed for the classical automotive control applications supporting the certification of the highest safety requirements. In contrast to this, the R-Car H3 features extensive video/image processing as well as multiple outputs for displaying a HMI. Although both hardware architectures claim ADAS as a use-case, the covered tasks inside are still completely different. On the one hand, the second generation AURIX provides a special hardware module in order to interface laser scanner or short distance radar. Therefore, the ADAS use-case includes data acquisition and low level data handling. The R-Car on the other hand, provides a powerful graphic processing module which can perform computation extensive operations, also including visualization.

	CPUs	Memory	Interconnects	Safety	use-case
AURIX	2/3x TriCore	<1 MB int. RAM ≤8 MB int. Flash	4x LIN 6x CAN/CAN-FD 2x FlexRay 1x Ethernet	Lockstep MPU ASIL-D	Gateway Domaincontroller Enginecontrol Airbag ADAS
R-Car H3	4x Cortex-A57 4x Cortex-A53 2x Cortex-R7	<1 MB int. RAM ≤8 GB ext. RAM ext. Flash	2x CAN/CAN-FD 1x MOST 1x Ethernet AVB	Lockstep MPU/MMU ASIL-B	Headunit Infotainment HMI ADAS

Table 2.1: SoC comparison table

Nevertheless, the R-Car H3 will never replace the AURIX within the EEA and vice versa. Both platforms have a reason for existence. The comparatively large computing power of the R-Car H8 is achieved by the use of performance optimized hardware like the integrated graphic processor or the Cortex-A clusters. The drawback at this point is, that the computation power is achieved with general purpose CPU designs, implementing techniques like speculative execution, branch prediction or out-of-order execution. Not only worsens those techniques the analysability and predictability, it also opens an attack space for novel exploitation techniques [75]. Therefore, the R-Car H3 provides lots of computation power for applications with limited safety requirements. In contrast to this, the AURIX family provides limited computation power for applications with the highest safety requirements.

2.2 Software Architecture

Modern automotive software is often developed and integrated according to the AUTOSAR [2] set of standards. This is especially the case for ECUs covered by more classical μC architectures like the AURIX family. AUTOSAR is defined by a consortium consisting of different partners from the automotive industry, including automotive software/hardware suppliers and Original Equipment Manufacturers (OEMs) as well as different research facilities. Figure 2.4 shows a simplified model of the AUTOSAR software architecture. In general, it consists of three parts. First, the *AUTOSAR Software* which contains the applications. Second, the *AUTOSAR RTE* which abstracts inter- and intra-ECU communication to the application. And third the *AUTOSAR Basis Software (BSW)* which integrates driver, services and two abstraction layers.

Applications in modern automotive systems implement a car's entire behavior. Since a few years, the development process of those applications is model based. One of the most common frameworks is Matlab/Simulink [49]. Combining this with automatic code generation leads to an efficient application development

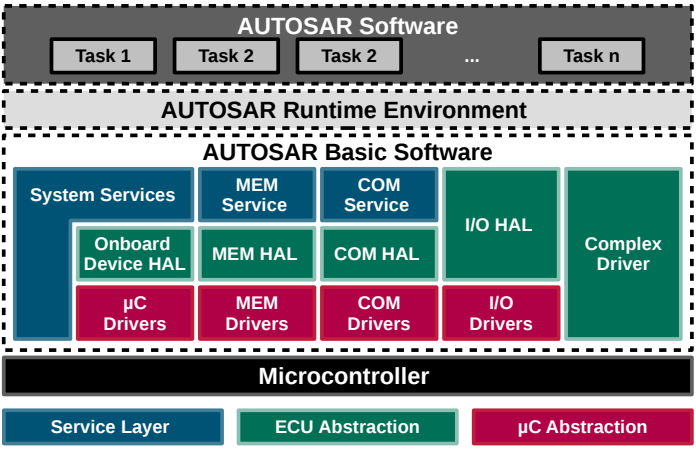


Figure 2.4: AUTOSAR Software architecture

process where source code can be generated after extensive simulation. Nevertheless, this does not describe how the application is later executed on an ECU. We will therefore give a brief introduction in the following Subsection 2.2.1.

The RTE decouples the high-level application from the ECU specific implementation of the BSW. One mechanism to achieve this is the AUTOSAR Virtual Functional Bus (VFB) [11], which abstracts the communication between different applications. Important at this point is, that it doesn't matter if an application reads/writes data from/to an application on the same or a remote ECU. The VFB either maps the access to the memory for internal communication or forwards it to the communication stack. Therefore, the VFB is described on a system level, while an actual RTE implementation on a single ECU just implements the local mapping. The data access from the application is then performed through so called ports, providing different communication paradigms further described in [11]. Although the VFB perfectly decouples the application from the underlying hardware, it is not a mandatory component of an ECUs software. Omitting VFB and RTE reduces the abstraction overhead, which is often the reason for this. As a detailed knowledge of the AUTOSAR RTE is not necessary to understand the mechanisms discussed afterwards, we refer to [11] for further information.

The BSW includes three different layers: the service layer, the ECU abstraction layer and the μC Abstraction Layer (MCAL). Implemented services range from a simple memory management, over communication up to the system services implementing the OS. As this dissertation deals with scheduling mechanisms, we describe the classic automotive OS further in Subsection 2.2.2. Both abstraction

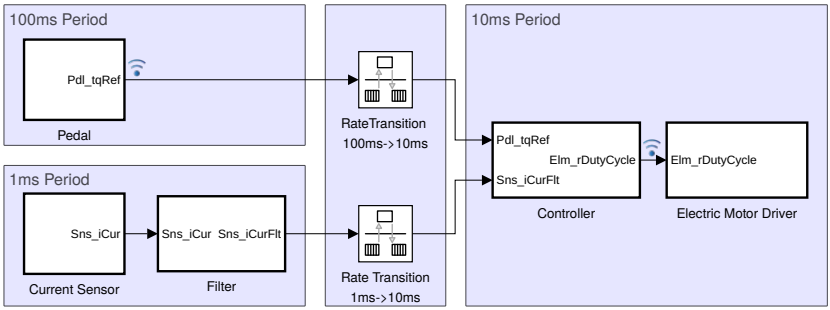


Figure 2.5: A simple engine controller in Matlab Simulink

layers implement drivers for either ECU or μC specific components. The implementation of an entire communication stack is used as an example to explain this further in Subsection 2.2.3. Also, the temporal behavior of a communication stack is important when considering scheduling techniques, as it is usually driven by interrupts and the application.

2.2.1 Application

Modern cars implement most of their important functionality as applications on several ECUs. Over the past decades remits in the automotive industry therefore moved from pure mechanical engineering to a combination of mechanical, electrical and control engineering, as well as computer science. As already mentioned, the function development is often based on frameworks like Matlab Simulink [49] enabling Model Based Design (MBD) [103]. MBD allows an abstraction to a visual functional description. Especially in case of control engineering MBD in combination with Matlab/Simulink is well-known, as it allows instant testing based on simulation. Combining MBD with automatic code generation leads to an efficient application development process. Especially if the generated code is already compliant to the relevant safety standards [43](e.g. ISO26262).

Applications in the automotive industry are usually developed as a set of functions. In the terminology of Simulink, those functions are also called *atomic subsystems*. An example for an application developed with atomic subsystem in Simulink is represented in Figure 2.5, showing a simple control for an electric motor. First, the throttle pedal is used as a reference input specifying a torque setpoint. Second, the controller calculates the corresponding PWM duty cycle, which should be set by the motor driver. And third, the actual current is measured and filtered in order to derive the actual torque at the drive shaft. Figure 2.5

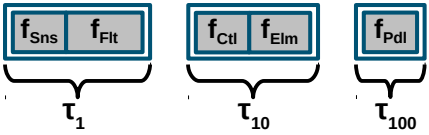


Figure 2.6: Possible runnable mapping for Figure 2.5

also shows the different periods (or rates) with which the corresponding functions are executed. In the context of an automotive embedded system, such a function combination with a defined dataflow is called *effect chain*.

A well-established period for a torque based controller is $10ms$ [59]. In contrast to this, a user defined input like the throttle pedal is executed with a greater period (e.g. $100ms$). Usually this is sufficient, as the fastest changes for the torque setpoint are based on human limitations. On the other hand, the actual current value is provided with a smaller period (e.g. $1ms$). This is often the case as higher sampling frequencies on such inputs allow more comprehensive digital signal processing which is cheaper compared to a pre-processing on the analog signals. Also, it might be the case that another application also uses the sampled current value and is executed with a different period. In order to combine the different periods in the model, Simulink uses *rate transition* blocks which provide data integrity during transfers between different periods (rates).

Generating code on a Simulink model can result in different high-level languages for software or hardware description. One of the most common ways in the automotive domain is the code generation to C [71] under the consideration of different development guidelines [85] and safety standards [66]. The example from Figure 2.5 would generate a C-function per atomic subsystem. Within the automotive domain those generated C-functions are called *runnables*. Those runnables are then grouped by common periods into *container tasks*. The signals between the different atomic subsystems like *Pdl_tqRef* or *Sns_iCur* result in global variables. This way a publisher subscriber based communication is implemented where variables are written only by one runnable, but can be read by several other runnables.

For the example from Figure 2.5, Figure 2.6 shows a possible set of generated runnabels. A container task therefore contains multiple runnables which are executed in a static order, determined by the dataflow of the application. In the automotive domain the preferred task scheduling technique implemented by the OS is preemptive and based on static priorities [13, 92], often mentioned as Static Priority Preemptive (SPP). Based on Figure 2.5 τ_{100} contains the processing of the pedal input (f_{Pdl}). The primary control algorithm (f_{Ctl}) and the output driver (f_{Elm}) are executed in τ_{10}. The current feedback including sensor input

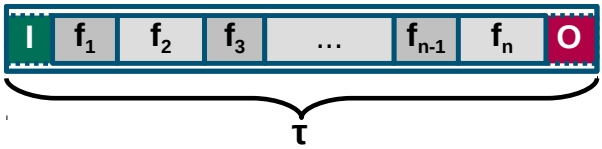


Figure 2.7: Generic container task with n runnables and in/output processing

(f_{Sns}) and filtering (f_{Flt}) is executed in τ_1. This set of three periodic tasks could now be scheduled by an OS executing the generated code from the Simulink model. τ_1 would be executed each millisecond, τ_{10} every ten millisecond and τ_{100} every hundred milliseconds. The priority assignment is done according to [80], assigning the highest priority to the task with the smallest period and vice versa. With a deadline equal to the task period, this is called Rate Monotonic Scheduling (RMS) and the most common type of scheduling (it implies SPP) in the classic automotive domain. In general, task periods in the automotive domain are usually in the range of $1ms - 500ms$. Sometimes even bigger periods are used for background activities.

Usually an ECU implements more than one single functionality based on several Simulink models. As a result, a container task usually contains runnables from multiple different applications. A more generic example for such a task with n runnables is shown in Figure 2.7. In addition to the runnables, the depicted task τ also contains data pre- (I) and post-processing (O) often used for scaling and local copies of global data. As already mentioned, Figure 2.6 shows a possible mapping of the generated runnables to container tasks. In some cases it might make sense to include f_{Pdl} in τ_{10}. One possible reason for a such a measure would be if the $100ms$ task would only contain a single runnable. To achieve a period of $100ms$ inside the $10ms$ f_{Pdl} would be executed conditionally based on a counter and a modulo operation. This way the number of tasks could be reduced, resulting in a lesser management overhead. Such an example is shown in Listing 2.1

The $10ms$ task is represented as *task_t10* and the $1ms$ task as *task_t1*. For both tasks, the contained runnables are executed in the shown order. The runnable for f_{Pdl} is executed every tenth task activation based on the counter *T10_ActCtr*. The function *IncT10_ActCtr* increments *T10_ActCtr* as an atomic operation and implements a task specific wrap around at the end of *task_t10*. In addition to that, both tasks also include calls to task specific pre- and post-processing functions (e.g. lines 12, 17, 23 and 31). The global variables in lines 2-5 implement the shown Simulink signals between the atomic subsystems and are accessed by the runnables according to Figure 2.5.

Listing 2.1: Possible C-representation of the example from Figure 2.5

```
/* Global variables */
static unsigend int Pdl_tqRef       = 0;
static unsigend int Elm_rDutyCycle  = 0;
static unsigend int Sns_iCur        = 0;
static unsigend int Sns_iCurFlt     = 0;

/* Task activation counter */
static unsigend int T10_ActCtr      = 0;
static unsigend int T1_ActCtr       = 0;

void task_t1(void)
{
  t1_data_pre();            /* Input data processing               */
  f_1_t1();                 /* Execute something else              */
  f_Sns();                  /* Execute sensor input                */
  f_Flt();                  /* Execute filtering                   */
  f_n_t1();                 /* Execute something else              */
  t1_data_post();           /* Output data processing              */
  IncT1_ActCtr();           /* Increment with overflow handling    */
  return;
}

void task_t10(void)
{
  t10_data_pre();           /* Input data processing               */
  f_1_t10();                /* Execute something else              */
  if(T10_ActCtr%10 == 0){   /* 100 ms "task" based on a counter    */
    f_Pdl();                /* Execute pedal input                 */
  }
  f_Ctl();                  /* Execute controller                  */
  f_Elm();                  /* Execute electric motor driver       */
  f_n_t10();                /* Execute something else              */
  t10_data_post();          /* Output data processing              */
  IncT10_ActCtr();          /* Increment with overflow handling    */
  return;
}
```

For the sake of clarity the following list shows the main characteristics of classic automotive applications.

- MBD is used for application development
- Code generation results in a set of runnables for each application
- Publisher subscriber communication between runnables based on global variables
- Container tasks
  - execute generated runnables in a static order
  - may include runnables of different applications but with common periods
  - repeated periodically
  - may use counter for multiple periods inside container tasks
  - implement data pre- and post-processing
- SPP task scheduling with priority assignment according to RMS
- Commonly used task periods in the range of $1ms - 500ms$

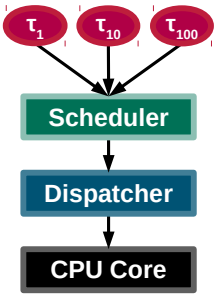


Figure 2.8: Task states in an AUTOSAR conform OS

2.2.2 Operating System

An OS is usually responsible for different actions. In general purpose computing the OS often provides multiple components including filesystems, driver and extensive resource management. In the AUTOSAR context, the operating system is considered a system service, which only provides minimal functionality. An OS conform to the Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK) or AUTOSAR OS definition [13] can therefore be considered as a microkernel [55, 53] design. [39] gives a good overview of the OSEK definition and the derived AUTOSAR OS. Primary tasks of an automotive OS are:

1. Task management
    - Scheduling
    - Context switching
    - Memory Protection
    - Conformance Classes
    - Communication
    - Synchronization
2. IRQ handling
    - Interrupt Service Routine (ISR) types
    - Alarm and counter management

The term *task management* bundles most of the important functions of an OS. First, the OS implements a scheduler, which determines the next task to be dispatched. This is shown in Figure 2.8 with three different tasks. The scheduler

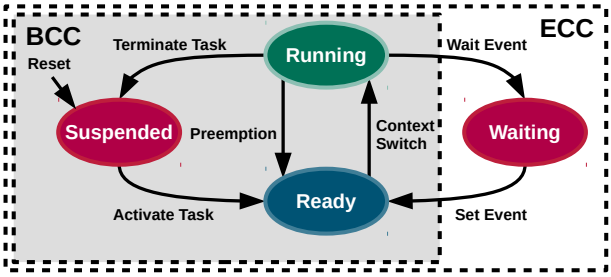


Figure 2.9: Task states in an AUTOSAR conform OS

implements a defined strategy, in order to make scheduling decisions. In case of an automotive OS, this scheduling strategy is based on fixed task priorities with a preemptive behavior as already mentioned. Although OSEK and AUTOSAR both dictate fixed priorities, systems like ERIKA OS [51] provide multiple scheduling mechanisms. Beside the standard automotive fixed priority scheduling, ERIKA OS also supports strategies like Earliest Deadline First (EDF) for research purpose. This is achieved, as the interface towards the dispatcher is always identical. Based on the scheduling strategy the scheduler hands over a task context to the dispatcher. After the scheduling decision, the following actions are independent of the implemented scheduling mechanism. If the task context is different from the actual context executed on the underlying core, the dispatcher issues a context switch. Although context switch routines are highly architecture dependent, most common steps are:

1. Save current task context on actual stack
2. Get new stack of the task to be scheduled
3. Restore previous (or initial) context from new stack

Often, an OS uses a so called Task Control Block (TCB) per task to store different task relevant parameters and runtime variables. As an example, such parameters can be scheduling dependent entries like the priority or the relative deadline (in case of EDF). OS dependent entries like a pointer to the current top of stack in case of a preempted task are also saved in the TCB. Additionally, the TCB might include a pointer, referencing a table with entries of available memory regions and access rights. Reconfiguring the MPU inside the context switch allows this way a memory protection on a task-level. The MPU reconfiguration would take place before the new task context is restored.

Another important TCB entry is the current task state. The number of different task states in an AUTOSAR/OSEK conform OS depend on the task con-

Listing 2.2: Example implementation of an BCC task

```
#define T10 1                /* Define task ID                        */

TASK(T10)                    /* Macro based function definition       */
{
  task_t10();                /* Execute generated container task      */
  TerminateTask();           /* Terminate task execution and call OS  */
  return;
}
```

formance class. Both, AUTOSAR and OSEK support two different conformance classes for tasks. So called Basic Conformance Class (BCC) tasks for execution without synchronization in between and Extended Conformance Class (ECC) tasks allowing synchronization during execution. In addition, both conformance classes can implement two different types of tasks. *Type 1*, allowing only one activation at a time per task with one task per priority and *Type 2*, allowing multiple queued activations at a time per task with multiple tasks per priority. Overall this leads to four different configurations based on the different conformance classes and different types (BCC1, BCC2, ECC1 and ECC2).

For the different task states only the conformance classes are relevant and not the different types. Figure 2.9 shows the different states for BCC and ECC tasks. The classic automotive system does not create tasks during runtime, but uses a statically defined setup. Therefore ,all available tasks are known at system startup and each task starts in the *Suspended* state after a system reset. A task acitivation switches the corresponding task to the *Ready* state, indicating that this task can be executed. Also, scheduling decisions are only performed on tasks in the *Ready* state. Next is the *Running* state which indicates the actual execution of the corresponding task. It is reached when a ready task is taken by the scheduler for execution and a context switch is performed through the dispatcher. Important at this point is, while several tasks might be ready for executing, only one task per CPU core can be in the *Running* state at a time. Which task is executed and therefore holds the *Running* state depends on the implemented scheduling strategy. Therefore, in case of SPP always the task with the highest priority of all tasks in the *Ready* state is executed.

In case of an BCC task, the *Running* state can be left in two different ways, first based on a finished execution and second based on a preemption. For the finished execution the task terminates itself and the task state is set to *Suspended* until the next task activation. In case of a preemption, the task is switched back to the *Ready* state. This might be the case, if a task with a higher priority enters the *Ready* state and is directly executed according to the scheduling strategy. The previously executed task will then continue execution, until it is the task with the highest priority in the system again. Listing 2.2 shows an example for a simple BCC task consisting of the container task function *task_t10* from Listing 2.1 and the task termination at the end.

Listing 2.3: Example implementation of an ECC task

```
#define T10 1                 /* Define task ID                        */
#define EV_T10 1              /* Define event ID                       */

TASK(T10)                     /* Macro based function definition       */
{
  EventMaskType mask;
  task_t10();                 /* Execute generated container task      */
  WaitEvent(EV_T10);          /* Wait for EV_T10 (enter waiting state) */
  GetEvent(T10, &mask);       /* Resume execution, get event for T10   */
  ClearEvent(EV_T10);         /* Clear event                           */
  do_something_else();        /* Do something else                     */
  TerminateTask();            /* Terminate task execution and call OS  */
  return;
}
```

In case of an ECC task, there is a third way to leave the *Running* state. As already mentioned, ECC tasks allow synchronization during execution. This synchronization is reached through so called *events* which can be set by several reasons. In order to enable synchronization, ECC tasks implement an additional state called *Waiting*, which is entered when a task waits for an event for synchronization. When the corresponding event is set, the task enters the *Ready* state again. Listing 2.3 shows an example for an ECC task consisting of the container task function *task_t10*, waiting for the event *EV_T10*, an additional function call and the task termination at the end.

Setting an event can have different reasons like an incoming message, synchronization with another task or also related to the underlying hardware through an IRQ. *IRQ handling* in general is grouped by AUTOSAR and OSEK in two different categories of ISRs. The first type *ISR1* is usually not directly related to the application, as it does not allow access to the corresponding functions inside the OS. Instead, it is usually used for low level driver handling inside the BSW. As an ISR1 does not provide any protection mechanisms, the switch to the IRQ context and back to the previous context introduces only minor runtime overhead. The lack of protection is the reason for the missing functionality for interaction with the application inside an ISR1. For this purpose there is the second type called *ISR2* which allows interaction with the application. This means, that e.g. from an ISR2 a task can be activated or also an event can be set, which results in a direct control of the application from the IRQ context. This is achieved with a more complex switch to the IRQ context resulting in more overhead compared to an ISR1. Beside the different ISRs, OSEK and AUTOSAR OS also support counter based alarms for single or recurring events. Like the entire operating system, alarms are configured during design time and invoke either a task activation, set an event or call a simple callback function. For more information regarding access rights of different ISR types and alarm callbacks, we highly refer to figure 12-1 from [92] or table 1 from [13].

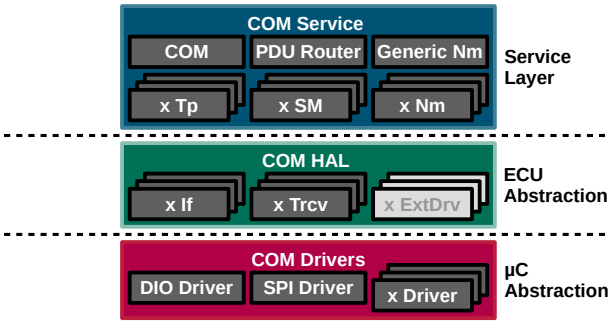


Figure 2.10: Software modules of an AUTOSAR conform COMStack

2.2.3 Software Stacks

Another important part of an automotive software architecture are software stacks. Software stacks in the context of AUTOSAR implement a set of modules vertically through all three layers of the AUTOSAR BSW. Considering Figure 2.4 two major stacks can directly be identified. First the Communication Stack (COMStack) including *COM Service*, *COM Hardware Abstraction Layer (HAL)* and *COM Drivers*, and second the Memory Stack (MEMStack) including the corresponding counterparts. Figure 2.10 shows the parts of an AUTOSAR conform COMStack in more detail. The specification contains several types for in-vehicle networks and buses. Most common communication media types are the already highlighted technologies like, CAN, LIN, FlexRay and Ethernet. The modules in Figure 2.10 starting with an *x* represent multiple modules of the same type for the different communication media. Be aware that not all available COMStack modules are shown in Figure 2.10 for complexity reasons. As an example, Figure 2.10 does not show higher layer protocols like Universal Measurement and Calibration Protocol (XCP) or Scalable Service-Oriented Middleware over Internet Protocol (IP) (SOME/IP), although those are part of the AUTOSAR COMStack definition.

Starting with the service layer, the primary interface towards the application or RTE is the *COM* module. From there on, the transmitted or received values are mentioned as Protocol Data Units (PDUs). We will use the transmit path in order to explain the functionality of the different modules. Inside the *COM* module, a multiplexing is also possible, such that a PDU might contain multiple different values. From there the PDU is passed to the *PDU Router*, which determines the corresponding communication medium used for the final transmission. This means, that the *PDU Router* passes the PDU to the corresponding

transport layer module *x Tp*. Inside *x Tp* the possible segmentation of a PDU is handled, based on the physical payload size of the underlying communication medium. Additionally, the service layer includes a medium dependent state management (*x SM*) as well as a network management (*Generic Nm*, *x Nm*). The state management is primarily used for stuff like flow control, while the network management is responsible for protocol specific features like a bus-sleep mode.

The ECU abstraction layer primarily include the interface (*x If*) and the transceiver modules (*x Trcv*) for each communication medium. From the corresponding *x Tp* module in the service layer, an outgoing PDU is passed to specific interface module. Each *x If* represents an abstraction of the underlying hardware to the upper service layer. This includes validity checks, transmit/reception confirmations as well as cyclic polling of underlying hardware through the corresponding *x Driver*. Each transceiver module is responsible for the configuration of hardware modules which represent the connection to the physical layer of the communication medium.

The μC abstraction layer includes the needed hardware drivers (*x Driver*) to access the communication peripherals for data link handling. The used transceivers for physical access of the communication media are usually off-chip and not part of the μC or SoC. Therefore, the μC abstraction layer also provides drivers for digital in/output (*DIO Driver*) or the often used Serial Peripheral Interface (SPI) (*SPI Driver*) in order to configure the off-chip transceivers. If the μC does not provide a corresponding communication peripheral, it is possible to have both, physical access and data link handling, off-chip. In this case, both both off-chip components may be interfaced through either digital in/output or SPI. The peripheral driver for data link handling is then part of the ECU abstraction layer (*x ExtDrv*).

AUTOSAR provides a detailed specification for each module. As an example [12] provides the specification for the *COM* module. For each module AUTOSAR specifies an API including type and function definitions. The API functions can be grouped into different types.

First, functions relevant for module initialization and control. As an example, this contains *Init*/*DeInit* as well as *GetStatus* calls.

Second, a set of functions which define the API towards the upperlying modules. In case of the *COM* module, this is the interface towards application or RTE.

Third, a set of callback functions representing the interface to underlying modules. As an example for the *COM* module, the function *Com_RxIndication* is registered in each *x If* during initialization. This way, the *COM* module defines which function should be called when a PDU was received.

Fourth, scheduled functions. Most of the modules provide so called *MainFunctions* which get executed periodically from a task context. Beside the IRQ

and callback driven functionality of the COMStack, the *MainFunctions* perform housekeeping and periodic PDU processing. For this purpose, the COMStack *MainFunctions* are often implemented as a single task with a higher period than the application tasks.

From a timing perspective, a COMStack therefore introduces two different types of load to a system. On the one hand the IRQ driven processing often based on low level driver handling, and on the other hand, the periodically execution of the *MainFunctions*. For an application, both can be considered as interference as those COMStack modules are executed on a higher priority.

2.2.4 Adaptive AUTOSAR

The previously described application model and system architecture from Section 2.2 is well-known and has proven itself in recent years. Nevertheless, this software architecture does not fit the needs for all future use-cases. Features like highly automated driving as part of ADAS, car2x communication or partial updates over the air are one of the most important technology drivers of today's automotive industry [50]. Primarily, highly automated driving require more powerful ECUs for local sensor fusion and processing. But also, the connection to backend servers get more important in order to receive something like current or predicted traffic data for route planning [88]. This way a car gets more and more connected to its surrounding environment and the internet. As already shown in [84], this steady connection can be exploited if it isn't done properly. The existing procedure, where an ECUs software is only updated in a verified garage, is acceptable for systems which are not remotely accessible. Compared to this, in case of an always connected system updates must be applied as soon as possible. Otherwise, the car would be prone to security leaks, which indirectly could endanger the passengers safety (e.g. [84] pages 83-86). Patching security leaks remotely over the air without having the car taken to the garage, is therefore the way to go. Also, via an over the air update additional functionality can be enabled at later date.

Again, those mentioned features are not compatible to the existing set of standards. One big reason for this is, that the existing AUTOSAR standards don't allow reconfiguration or partial updates during runtime. Therefore, the AUTOSAR consortium decided to provide another set of standards called AUTOSAR Adaptive Platform (AP) and rename the existing specification to AUTOSAR Classic Platform (CP). Important is at that point, that the AP is not meant to be a replacement of the CP. Systems of both standards should be able to coexists side by side in the same EEA, on the same ECU or also on the same μC/SoC. Table 2.2 shows the main differences between both, CP and AP.

First of all, the OS of the AP is no longer based on theOSEK specifications.

	AUTOSAR CP	AUTOSAR AP
OS	OSEK	POSIX
Memory Management	Only MPU for safety	Virtual address space per application with an MMU
Code Execution	Statically from Read Only Memory (ROM)/Flash	Dynamically loaded to RAM during runtime
Task Scheduling	Fixed number of tasks, scheduling based on priority	Dynamic number of tasks, different scheduling mechanisms
Communication Protocols	Signal based (CAN,FlexRay etc)	Service oriented (SOME/IP)
Exemplary Hardware	AURIX	R-Car H3

Table 2.2: Comparison table of the AUTOSAR CP and AUTOSAR AP based on [21]

Instead, a minimal subset (PSE51 [14, 16]) of the POSIX standard [113] is used, defining an API between applications and OS. There are several OSs available, which are either certified or mostly compliant to the POSIX standard. Due to its open source format and wide hardware support, Linux [19] is one possible OS for the AUTOSAR AP [88]. Receiving updates over the air requires a MMU in order to provide multiple memory address spaces. Reason for this is the fact, that during compile time the physical address where the final binary is later placed, can not be determined. Instead, applications or updates are linked to a fixed and constant memory layout, which is later provided by the MMU. Each application is executed in its own virtual address space. As a result of this technique, applications are not executed from the ROM/Flash anymore, but loaded into the RAM beforehand. As task scheduling and configuration belongs to the operating system, the AP supports several dynamic mechanism defined by POSIX [118]. In general, an application is not stuck to a fixed number of tasks anymore. According to [14], the first supported scheduling algorithms are *SCHED_FIFO* and *SCHED_RR*, as defined by POSIX. Nevertheless, first measurements show that a Linux based AP struggles with reaching the same hard real-time behavior compared to a much slower CP [67]. The communication between different ECUs is limited to SOME/IP in case of an AP. As already mentioned, the AUTOSAR COMStack supports SOME/IP as a higher layer protocol. This way the communication between CP and AP ECUs is ensured. Also, this way the AP gets access to the existing in vehicle network. We already mentioned that the different hardware architectures explained in Section 2.1 are both relevant for modern ECUs. Both hardware architectures were chosen as examples for good reason. The AURIX is a well-known and widely used μC for ECUs based on the AUTOSAR CP.

The R-Car H3 as an SoC instead fits perfectly into the profile of an AUTOSAR AP ECU.

With the first definition of the AP, AUTOSAR also introduced the possibility of virtualization through a hypervisor [16]. The execution of multiple AP or even CP instances on the same SoC are therefore considered a possible integration by the AUTOSAR consortium. According to ISO26262, the hypervisor is then in charge of preserving freedom from interference between different AUTOSAR AP or CP instances. Although the standardization is in an early stage of progress at the moment, different software suppliers in the automotive industry are working on actual hypervisor implementations [88, 30]. As already mentioned, future hardware platforms like the R-Car H3 already provide hardware mechanisms to support virtualization.

2.3 Problem Statement

With the introduction of the AURIX μC family, the evolution of multicore CPUs reached the automotive industry. While the AURIX is primarily targeted to the CP, same applies to the newly developed AP with SoCs like the R-Car H3. For both platforms an efficient use of the available processing power is key to a good overall performance. Dependent on the platform type, the limiting constrains for efficient use of the available resources vary. On both AP and CP the parallel execution and synchronization among multiple cores of existing legacy software is a big issue and needs to be addressed. Also important is, the application mapping to different cores, as well as the freedom from interference between those applications. As an example, [86] proposes a mapping where on core does only contain periodically executed applications. Non periodic software like IRQ handling is then executed on another core. While this often can hardly be reached, it also isn't always useful to completely decouple periodic and sporadic execution on different cores due to latency issues.

2.3.1 Efficient Hypervisor Scheduling

The AP paves the way for much more powerful ECUs which can be used for either infotainment or also computing intense operations in ADAS. Nevertheless, the efficient use of such SoCs is still challenging, as parts of different applications need to be mapped to the available cores. Figure 2.11 shows a mapping based on an example described in [88]. The ECU in Figure 2.11 contains two different types of CPU cores. A set of performance cores and a set of safety cores. The R-Car H3 could be a possible SoC for such a setup with Cortex-A covering performance and Cortex-R safety tasks. The safety cores are used for system monitoring and therefore execute an AUTOSAR CP. Contrary to this, the performance cores are

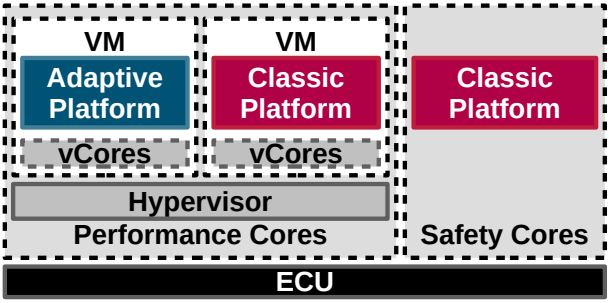


Figure 2.11: Mixed software mapping with AP and CP

under the control of a hypervisor executing two different AUTOSAR instances. In this mapping an AP would execute the primary application. As the AP only provides SOME/IP for communication, an additional CP is used to connect the ECU to the common interconnects like CAN, FlexRay or LIN.

In such a configuration the hypervisor provides virtual CPU cores (*vCores*) to the upper software layers. In other words, the hypervisor provides a Virtual Machine (VM) to each AUTOSAR instance. Each VM might contain multiple virtual cores. How the virtual cores are mapped to the available physical cores is managed by the hypervisor. Same applies to the translation between virtual and physical memory addresses. When multiple virtual cores are mapped to the same physical core, the hypervisor acts as a resource arbiter. From the perspective of the hypervisor, the different virtual cores can be interpreted as tasks scheduled on a physical core. Result of this technique is a hierarchical scheduling containing hypervisor level and OS level task scheduling. At this point it is important to mention that, even though the AP considers hypervisor based virtualization to be a possible way to integrate different applications on the same hardware, it does not provide any definition about the actual integration. Therefore, a formal definition on how the hypervisor level scheduling should be integrated is missing completely.

While being fairly new in the automotive domain, hierarchical scheduling is well-known in avionics as part of ARINC653 [93]. We already mentioned PikeOS [70] as an example for a well-established OS and virtualization environment in avionics. In order to provide temporal isolation among multiple VMs, ARINC653 uses a time partitioning with TDMA. This means, that partitions (VMs) get executed by the hypervisor cyclically in a fixed order for fixed amounts of time. As a result, each partition is executed for a fixed amount of time within a certain period, independent of the behavior of any other partition. While TDMA provides a perfect temporal isolation this way, the scheduling isn't work conserving. This

means, that the processor can be idle (because the current partition is idle), even though another partitions has outstanding workload. At this point processing power is wasted and response times of task or IRQ inside partitions rise. This entire behavior is explained further more in Chapter 4.

Since IRQs are an essential part of the existing autmotive software architecture for software stacks or applications, longer response time are particularly bad. In case of avionics, those problems can be solved with a faster and more expensive processor. For a high volume market like automotive, wasting processing power is not applicable. As a result, a new work conserving scheduling for the hypervisor is needed. In order to enable freedom from interference (as required by ISO26262), a sufficient degree of temporal isolation must be ensured.

Requirements

We address the previously described issue according hypervisor scheduling under the following requirements. The proposed mechanism should provide:

1. Work-conserving scheduling
2. Improved response times
3. Sufficient degree of temporal isolation
4. Low overhead implementation

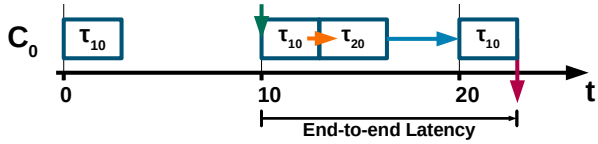
Proposed solution

In order to provide the required isolation as part of an efficient hypervisor scheduling, we propose a modification of the partition based scheduling described in ARINC653. Chapter 4 gives an introduction to ARINC653 and highlights first issues. Chapter 5 proposes our final scheduler modification based on the SPS mechanism. The implementation is then extensively explained in Section 7.1.

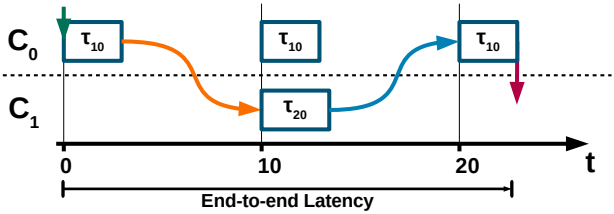
2.3.2 Multicore Synchronization

As already mentioned, a major problem during the migration to multicore ECUs is the efficient usage of the additional cores for existing legacy software. Reason for this is the fact, that most of the software has been developed for singlecore ECUs. Often, single software parts (e.g. runnables) have proven themselves and were not been touched the past years. Tearing them apart and distributing them to different cores can cause major problems due to data inconsistencies. What's surprising is that less interference by other software parts can be a problem.

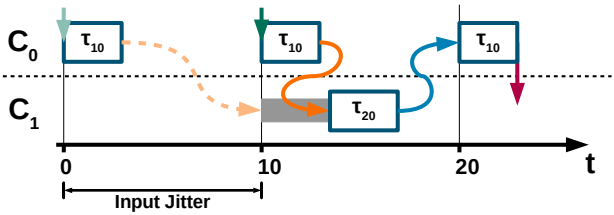
Figure 2.12 provides an example for clarification. Let us assume an effect chain, where input data (green arrow) is first processed by τ_{10}. The intermediate



(a) Singlecore implementation



(b) Multicore implementation without interference



(c) Multicore implementation with interference

Figure 2.12: Task distribution from single- to multicore

data (orange arrow) is then passed to τ_{20} where the final data (blue arrow) is generated. For this final data, the output (red arrow) is then generated in τ_{10}. Such a setup is very common in the automotive domain. τ_{10} can be interpreted as the periodic COMStack task which connects the application tasks to the in-vehicle network. The application is then implemented as part of τ_{20}.

Figure 2.12a shows this for a simple singlecore (C_0) implementation. τ_{10} is activated and executed every $10ms$. Same applies to τ_{20} every $20ms$ starting with an offset at $t = 10ms$. Due to the priority assignment based on RMS, τ_{10} has a higher priority than τ_{20}. Therefore, τ_{10} is always executed first, even though τ_{20} was activated at the same time. This way the RMS-based priority assignment supports the desired dataflow, as both tasks are scheduled and executed on the same core. The time between processing the first input data (green arrow) and producing the final result (red arrow) is mentioned as an effect chains *End-to-end Latency*.

On a system with multiple cores this might not be the case anymore. Figure 2.12b shows an example where τ_{10} and τ_{20} are mapped on different cores. A reason for this might be the attempt to isolate the COMStack on a distinct core. This way the interference to the application caused by IRQs and the periodically executed *MainFunctions* would be removed. While the priority based scheduling delayed the execution of τ_{20} in the singlecore implementation, this is not the case for the shown multicore implementation. Without the interference from τ_{10}, τ_{20} starts execution immediately at $t = 10$ on core C_1. As τ_{10} is executing in parallel on C_0, τ_{20} will read its input value before the newest value was written by the current instance of τ_{10}. Instead, τ_{20} reads the value written by the previous instance of τ_{10} activated at $t = 0$. This way the application in τ_{20} is working on older input data, which directly increases the end-to-end latency.

In the provided example in Figure 2.12b τ_{20} is working on older input data compared to the singlecore implementation from Figure 2.12a. If such an input delay is acceptable or not depends on the underlying application software. Also, methods like model based prediction can be used if the input delay is constant. However, a problem arises if the input delay is not constant. As an example Figure 2.12c shows the same task mapping as Figure 2.12b but introduces additional interference on C_1. Reason for the additional interference can be another task with a higher priority, IRQ handling or an application task with Variable Rate-Dependent Behavior (VRB) [38]. Such VRB tasks can be found in ECUs for combustion engine control, where at least one task is activated based on the rotation of the crankshaft. Due to the interference the execution of τ_{20} is delayed in such a way that it starts after τ_{10} has finished execution on C_0. This way τ_{20} reads the most recent value like in Figure 2.12a resulting in the same end-to-end latency. In case of a sporadic interference, the execution of τ_{20} is not always delayed and therefore the previous input value is read (dashed orange arrow). As a result of this behavior, the input delay can not be considered as constant anymore because of the so called *input jitter*. The authors in [106] showed in a simple example that even a small input jitter might cause serious problems. In some cases it may even lead to an unwanted instability of the implemented control algorithm [90].

Even though the used example in Figure 2.12 shows a perfect example for the CP, same applies for the coordination between different cores in a virtualized environment. Therefore, the same behavior can be expected when executing entire virtualized partitions on different cores without synchronization, as part of an AP integrated system.

Requirements

We address the previously described issue according synchronization of legacy software across multiple cores, under the following requirements. The proposed mechanism should provide:

1. Comparable end-to-end latency relation to singlecore
2. Minimized input jitter
3. Low overhead implementation

Proposed solution

In order to address the previous requirements, we take a look in this dissertation at the LET paradigm. Chapter 6 provides a basic explanation as well as the theoretical foundation. The implementation is then extensively explained in Section 7.2.

"So, in the face of overwhelming odds, I'm left with only one option: I'm going to have to science the shit out of this."

- Mark Watney

CHAPTER 3

System Model

This chapter presents the used system model for the underlying RTA. We provide the formal foundation for both challenges from Section 2.3. But, in order not to anticipate too much, mechanism specific system attributes will be introduced in the corresponding chapter. Even though the RTA for the proposed mechanisms uses the same framework, there are, however, significant differences. This chapter uses therefore more generic definitions, which fit the proposed mechanisms of both challenges from Section 2.3.

In general the topic of extensive RTA has been well-known for years. In order to guarantee real-time properties suitable frameworks are Compositional Performance Analysis (CPA) [58] or Real-Time Calculus (RTC) [114]. As a part of Luxsoft, Symtavision provides with SymTA/S a commercial CPA implementation [81]. Beside SymTA/S, Python Implementation of Compositional Performance Analysis (pyCPA) [40] provides an open source CPA implementation. Within this dissertation we use CPA as a framework in order to show the formal benefits of our mechanisms. The used system model is based on [110, 17, 87]. While CPA is capable of system level analysis, we focus on the local resource analysis. Our model is therefore slightly modified and reduced compared to the one used in [110, 17, 87]. We start with a structural platform definition, provide an execution model and explain the fixed-point iteration of the used RTA.

Definition 3.1: Platform

A platform $\mathcal{P}$ is defined as a directed graph

$$\mathcal{P} = \langle \mathcal{R}, \mathcal{E} \rangle \tag{3.1}$$

with $\mathcal{R}$ as vertices, representing a set of m resources $\mathcal{R} = \{\phi_1 \dots \phi_m\}$. $\mathcal{E} \subseteq \{(\phi_a, \phi_b) : \phi_a \neq \phi_b \in \mathcal{R}\}$ are the edges of the directed graph which define connections between different resources.

Such resources might be a CPU core, an interconnect or also an interface to a shared memory. Within the context of this dissertation, we consider only CPU cores as resources.

Definition 3.2: Resource

Each resource $\phi_i \in \mathcal{R}$ is defined as

$$\phi_i = \{\Gamma_i, \mathcal{S}_i, b_i(t)\} \tag{3.2}$$

with $\Gamma_i = \{\tau_1 \dots \tau_m\}$ as s set of m tasks, $\mathcal{S}_i$ as scheduler for resource arbitration and $b_i(t)$ as service function.

3.1 Execution Model

In general, a resource provides service to a set of tasks. Service allocation to the different tasks is managed by the implemented scheduling strategy. The execution model defines, how the provided service is consumed by the tasks. As already mentioned in Section 2.2.1, tasks implement the actual application. The following parameters can therefore directly be derived from the corresponding application.

Within the domain of real-time systems usually the average-case performance is negligible. More important instead are bounds for best- and worst-case behavior. The following definitions are therefore tailored to fit the best-/worst-case analysis.

Definition 3.3: Task

A task τ_i is defined as

$$\tau_i = \{\mathcal{ET}_i, \mathcal{EM}_i, \mathcal{TP}_i, D_i\} \tag{3.3}$$

with $\mathcal{ET}_i$ defining execution time bounds, $\mathcal{EM}_i$ as event model, $\mathcal{TP}_i$ as generic task parameter and D_i as relative deadline.

The generic task parameter depends on the used scheduling strategy. As an example, in case of a priority driven scheduling it holds the task priority. For a time driven scheduling like TDMA the parameter would represent the timeslot size.

In order to perform any kind of action or calculation, a task needs to consume service on the corresponding resource. In case of CPU cores, this received service is called execution time. Due to the behavior of complex software, this execution time is often not constant for consecutive activations of the same task. One reason for this is conditional execution based on if-statements (e.g. Listing 2.1 line 27). The execution time model therefore provides bounds for minima and maxima.

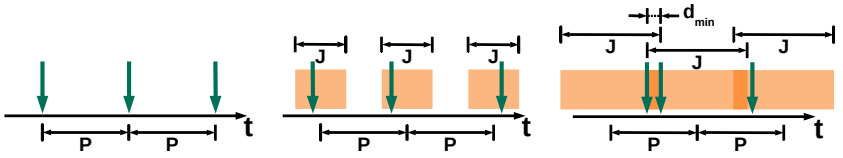
Definition 3.4: Execution time

The execution time model $\mathcal{ET}_i$ is a tuple with two entries

$$\mathcal{ET}_i = \{\underline{C}_i, \overline{C}_i\} \tag{3.4}$$

where $\underline{C}_i$ is the best-case (Best-Case Execution Time (BCET)) and $\overline{C}_i$ the Worst-Case Execution Time (WCET) of τ_i.

Beside the different execution times, we also need to model task activations. A common method is to describe multiple task activations as a stream of events, where each event corresponds to one task activation. Within the time domain this stream has an infinite length. An event model uses a set of parameters to describe the behavior of such a stream in a formally. As an example, Figure 3.1a shows an event stream for a periodically activated task. Each perpendicular arrow indicates an event and therefore a task activation. The parameter P defines the period used for event triggering. In order to provide a more versatile model, an additional jitter J can be introduced allowing an event to occur with a specified region. This is shown in Figure 3.1b. In case of $J \geq P$, the specified regions in which an event can occur per period overlap. In order to avoid multiple events at the same time a minimum distance d_{min} between two consecutive events can be defined as shown in Figure 3.1c. This way also bursts of multiple events can be



(a) Periodic (b) Periodic with jitter (c) Periodic with jitter and minimum distance

Figure 3.1: P, PJ and PJd event model

modeled. The three shown examples are named according to their parameters as P, PJ or PJd event model. Also, [99] describes several other event models like sporadic or sporadically periodic events.

Definition 3.5: Event model

The $\mathcal{EM}_i$ of τ_i is given as

$$\mathcal{EM}_i = \{\mathcal{EP}_i, \eta_i, \delta_i\} \tag{3.5}$$

where $\mathcal{EP}_i$ is a set of parameters describing $\mathcal{EM}_i$ in the time domain. η_i and δ_i are derived sets of functions, describing bounds on $\mathcal{EM}_i$ in the time window domain.

While $\mathcal{EP}_i$ represents the event model's parameter set, additional functions for event bounds are needed for the later discussed RTA. Instead of an infinite event stream, the RTA only considers time windows starting at the so called *critical instant* [80] .

Definition 3.6: Critical instant

The critical instant defines that the worst-case interference to τ_i is caused, when all interfering tasks are activated at the same time as late as possible according to their event model. All subsequent activations arrive then as early as possible. According to [80] this results in the worst-case response time of τ_i.

According to the critical instant, arrival and distance functions are then defined as:

Definition 3.7: Arrival functions

η_i is a set of arrival functions

$$\eta_i = \{\eta_i^-(\Delta t), \eta_i^+(\Delta t)\} \tag{3.6}$$

where $\eta_i^-(\Delta t)$ returns the minimum number and $\eta_i^+(\Delta t)$ the maximum number of events for τ_i that can arrive within any time window of size Δt relative to the critical instant.

Definition 3.8: Distance functions

δ_i is a set of distance functions

$$\delta_i = \{\delta_i^-(q), \delta_i^+(q)\} \tag{3.7}$$

where $\delta_i^-(q)$ returns the minimum distance and $\delta_i^+(q)$ the maximum between the first and the $q'th$ consecutive event of τ_i, where the first event occur at the critical instant.

The distance functions are therefore a dual representation of the arrival functions. E.g. η_i^+ corresponds to δ_i^-, while η_i^- corresponds to δ_i^+ and vice versa.

Both, arrival and distance functions can be derived from $\mathcal{EP}_i$. As an example, for the shown PJ model from Figure 3.1b $\mathcal{EP}_i$ is given as:

$$\mathcal{EP}_i = \{P_i, J_i\} \tag{3.8}$$

Under consideration of the critical instant, the corresponding arrival and distance functions are derived to:

$$\eta_i^-(\Delta t) = \left\lfloor \frac{\Delta t + J}{P} \right\rfloor, \qquad \eta_i^+(\Delta t) = \left\lceil \frac{\Delta t + J}{P} \right\rceil \tag{3.9}$$

$$\delta_i^-(q) = (q-1) \cdot P - J, \qquad \delta_i^+(q) = (q-1) \cdot P + J \tag{3.10}$$

As [99] provides a detailed explanation how, either arrival or distance functions can be constructed based on the corresponding event model parameter set, we won't discuss this step in further detail. In order to provide a more elegant notation, we define $\mathcal{EM}^P(P)$, $\mathcal{EM}^{PJ}(P, J)$ and $\mathcal{EM}^{PJd_{min}}(P, J, d_{min})$ as functions returning a corresponding event model for a P, PJ or PJd task.

One big difference between real-time and none real-time systems is the existence of an actual task deadline.

Definition 3.9: Relative deadline

The deadline D_i defines the latest possible time at which an activation of τ_i must be finished relative to the corresponding event, responsible for activating τ_i.

Without a deadline an important constraint would be missing, as it is crucial for the functional correctness of an application. As an example, the functional correctness of an airbag ECU is not only defined by the task of triggering the airbag. The task must be finished in a strictly defined period, as otherwise the passengers face might hit the steering wheel.

Knowing the execution model of a task, we can derive several additional definitions to characterize tasks, resource and scheduler further more. In some cases it makes sense to provide a rough estimation in order to estimate a task-sets schedulability.

Definition 3.10: Laxity

The laxity L_i of τ_i is defined as

$$L_i = D_i - \overline{C}_i \tag{3.11}$$

As shown above, the laxity depends only on the WCET and the relative deadline. Both parameters either depend on the resource and/or the application. The information value give by the laxity is torn. While $L_i < 0$ indicates that τ_i can not be scheduled on the corresponding resource, the opposite does not necessarily apply for $L_i \geq 0$. Important is, that the laxity does not consider any interference during execution. It is therefore independent of the underlying scheduler and the residual set of tasks on the resource.

Also, the current state of a resource is relevant for analysis as well as later for scheduler implementation. We therefore define two different states. The first is given as:

Definition 3.11: Busy

A resource ϕ_i is considered as busy as long as at least one of the tasks in Γ_i is processing an activation on ϕ_i.

With this definition we define the second state as:

Definition 3.12: Idle

A resource ϕ_i is considered as idle if it is not busy.

As already stated in Section 2.3.1, one objective of this dissertation is a work-conserving scheduling. For unambiguity, we define work-conserving based on Definition 3.12 as follows.

Definition 3.13: Work-conserving

A scheduler $\mathcal{S}_i$ is considered as work-conserving, when ϕ_i is only idle, if none of the tasks in Γ_i has an unfinished activation.

3.2 Response Time Analysis

With the execution model complete, we can now explain the actual RTA. [80] is often mentioned as one of the most popular and earliest publications concerning the schedulabilty of task-sets. The proposed theorems in [80] provide a simple estimation based on the utilization of the entire task-set. This was possible as the tasks were limited by a simple periodic event model with deadlines equal to their periods. We already mentioned this type of scheduling as RMS. As a result, the proposed theorems provide a simple boolean statement about the schedulablity of the task-set without giving any information about the actual task response times.

Definition 3.14: Response time

The response time is defined as the time between activation and termination of the task including all preemptions by other tasks. $\mathcal{RT}_i$ defines a tuple with two entries

$$\mathcal{RT}_i = \{\underline{R}_i, \overline{R}_i\} \tag{3.12}$$

where $\underline{R}_i$ is the best-case (Best-Case Response Time (BCRT)) and $\overline{R}_i$ the Worst-Case Response Time (WCRT) of τ_i.

A well-known method to compute WCRTs is the busy-window analysis introduced in [69]. General idea is to compute the time, the resource is busy processing an activation of the considered task and its interferes. In [69], this method was again limited to RMS with implicitly defined deadlines at the end of each period. Later in [78] the busy-window method (busy period in this paper) was modified in order to handle arbitrary deadlines. When considering arbitrary deadlines, queued task activations may occur. This might be the case for a task with periodic activations but $D > P$. In order to deal with this, [78] defined the level-i busy period, starting with the critical instant. The authors proofed that

the longest response time during that period corresponds to the WCRT. With the already mentioned arrival and distance functions, [104] modified the busy-window method in order to handle arbitrary task activation patterns. The result was the definition of the multiple event busy-window (busy-time in [104]). Important is at this point, that the busy-window method in general is only defined to be applicable on work-conserving schedulers. This also holds for the later defined multiple event busy-window.

Primarily the busy-window technique was developed for priority based scheduling like RMS or SPP. In [104] the multiple event busy-window $w_i(q)$ of the q'th was given as

$$w_i(q) = q \cdot \overline{C}_i + \sum_{\tau_j \in hp(i)} \eta_j^+(w_i(q)) \cdot \overline{C}_j \tag{3.13}$$

with $hp(i)$ as set of interfering tasks with a higher priority than τ_i. As $w_i(q)$ occur on both sides of the equation, $w_i(q)$ must be calculated several times

$$\begin{aligned} w_i^0(q) &= 0 \\ w_i^1(q) &= q \cdot \overline{C}_i + \sum_{\tau_j \in hp(i)} \eta_j^+(w_i^0(q)) \cdot \overline{C}_j \\ &\dots \\ w_i^n(q) &= q \cdot \overline{C}_i + \sum_{\tau_j \in hp(i)} \eta_j^+(w_i^{(n-1)}(q)) \cdot \overline{C}_j \end{aligned}$$

until a fixed-point is reached ($w_i^n(q) = w_i^{(n-1)}(q)$). While $q \cdot \overline{C}_i$ only depends on the analysed task, the sum on the right depends on the priority based scheduling strategy. In order to provide a more generic definition, we replace the sum with a simple blocking term. Within this dissertation we define the multiple event busy-window as follows:

Definition 3.15: Multiple event busy-window

The q-event busy-window $w_i(q)$ is given as the time needed to process q activations of τ_i,

$$w_i(q) = q \cdot \overline{C}_i + B_i(w_i(q)) \tag{3.14}$$

where $B_i(\Delta t)$ is the caused interference by other tasks on the same resource.

$B_i(w_i(q))$ is then defined for each scheduling strategy separately. To determine $\overline{R}_i$ of τ_i, the first Q_i activations of τ_i have to be considered.

$$Q_i = \max\left(n : \forall q \in \mathbb{N}^+, q \leq n : \delta_i^-(q) \leq w_i(q-1)\right) \tag{3.15}$$

A more intuitive description is, that the $(q+1)'th$ activation must be considered if it occurred during the busy-window of the first q activations [78]. The worst-case response time is then given as the maximum of all observed response times of the analysed task during the multiple event busy-window. The earliest $q'th$ activation relative to the critical instant is given through the minimum distance function δ_i^-. With the corresponding window size of the $q'th$ activation this leads to:

$$\overline{R}_i = \max_{q \in [1, Q_i]} \left(w_i(q) - \delta_i^-(q)\right) \tag{3.16}$$

With knowledge of the WCRT, we can define the schedulablity of an entire task-set.

Definition 3.16: Schedulability

A set of tasks Γ_i is considered as schedulable if all tasks have WCRTs lower or equal their relative deadlines

$$\Gamma_i schedulable \Leftrightarrow \forall \tau_j \in \Gamma_i : \overline{R}_j \leq D_j \tag{3.17}$$

Compared to the provided mechanisms in [80], a scheduleability definition based on actual WCRTs is more computationally intensive but independent of the used scheduling mechanism and not restricted to RMS.

We already introduced the laxity in Definition 3.10 and mentioned that additional interference is not taken into account. This is different in case of task slack.

Definition 3.17: Task slack

The slack S_i of τ_i is defined as

$$S_i = D_i - \overline{R}_i \tag{3.18}$$

Task slack is calculated based on WCRT and deadline. It therefore includes scheduling based interference by other tasks and provides an information about the flexibility of the task-set. In case of a larger task slack, the task can be delayed for a certain amount of time without missing its deadline. This can be

very useful since it allows subsequent changes. This knowledge is later used for system optimization through scheduling modifications.

"I'm a leaf on the wind. Watch how I soar!"

- Hoban Washburne

CHAPTER 4

ARINC653 based Hypervisor Scheduling

Since a few years virtualization has becomme more and more relevant, also within the world of embedded systems. Modern CPUs like the introduced R-Car H3 provide additional hardware in order to support virtualization efficiently.

As mentioned in Section 2.3.1 we consider ARINC653 a possible basis for the mechanisms needed to ensure freedom from interference between different applications on the same ECU. The remainder of this chapter shows, how ARINC653 achieves the mentioned temporal isolation among its virtualized partitions. We also highlight the drawbacks of this approach regarding IRQ and task response times. To mitigate these problems we provide a possible modification to the hypervisor scheduling and discuss its pros and cons. Most of the contribution in this chapter has been published before in [28, 23].

4.1 System model addition for hierarchical scheduling

The scheduling in ARINC653 is hierarchically structured based on TDMA and SPP. An example with two partitions (P_1 & P_2) is given in Figure 4.1. The hypervisor executes the different partitions after each other and repeats this cyclically. Inside a partition the tasks of the virtualized application are scheduled preemptive based on their priorities. A task can therefore be disturbed during execution

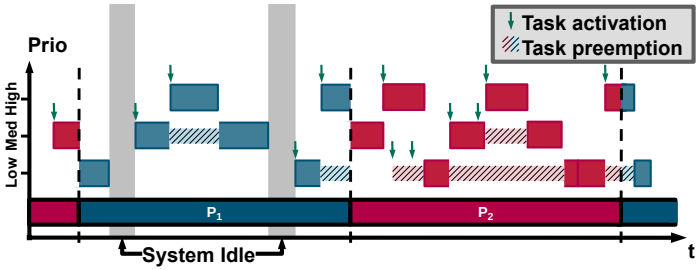


Figure 4.1: ARINC653 inspired scheduling

for several two reasons. First, it is preempted based on interference inside the partition caused by another task of the same application. And second, the partition itself is at the end of its timeslot and the hypervisor switches to the next partition in the TDMA schedule. Also, Figure 4.1 shows the major drawback of TDMA based scheduling. It is not work conserving. At two points during the execution of P_1 the system is idle, as none of the tasks in P_1 has an outstanding activation. In contrast to this, the medium priority task in P_2 would be ready to execute.

In order to describe such a hierarchical system formally, we need an addition to the system model described in Chapter 3. For the extended model a task-set Γ_p with multiple tasks is mapped to a partition P_p. A partition-set Γ_{HYP} with multiple partitions is then mapped to a resource. The number of partitions inside Γ_{HYP} is denoted by Ω. According to Definition 3.3 we need to adapt the partition parameters to the given task model. While each partition always contains an additional task-set for the POS, other parameters depend on the scheduling strategy used by the hypervisor. In case of TDMA each partition P_p has a timeslot with size T_p. The partitions are executed in their corresponding timeslots with a static order repeated every T_{TDMA}.

$$T_{TDMA} = \sum_{p \in \Gamma_{HYP}} T_p \tag{4.1}$$

As shown in Figure 4.1 the actual used time for task execution inside a partition doesn't have to be the same compared to the size of the corresponding timeslot. In order to handle this in formal way, we define $T_{p,min}$ as the minimum time needed during the timeslot T_p in order to finish all partition internal tasks before their deadlines. A result of this observation is the so called partition slack.

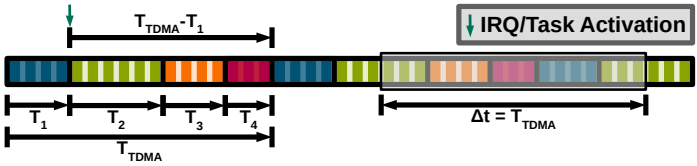


Figure 4.2: Temporal isolation in any time window of size T_{TDMA}

Definition 4.1: Partition slack

The partition slack $T_{p,S}$ is given as the time, which is not necessary needed by the task-set Γ_p inside in order to reach all deadlines.

$$T_{p,S} = T_p - T_{p,min} \tag{4.2}$$

A way how $T_{p,min}$ can be calculated is shown later in Section 4.5. With $T_{p,min}$ defined, we can now describe a partition under TDMA scheduling based on Definition 3.3. From the perspective of the hypervisor the caused interference by a partition P_p can be considered as a periodic task τ_p with period T_{TDMA} and T_p as execution time. In order to calculate the multiple event busy-window for a task τ_i inside a partition P_p, we simply rewrite the equation from Definition 3.15 to

$$w_{p,i}(q) = q \cdot \overline{C}_{p,i} + B_{p,i}(w_{p,i}(q)) \tag{4.3}$$

where $B_{p,i}(\Delta t)$ includes both, the blocking caused by the hypervisors and the partition internals scheduling strategy.

We already mentioned that temporal isolation is given, if the response time of a task is independent of the behavior of all other tasks on the same resource. While we can describe a partition as an interfering task, we can not directly define a response time for a partition. Another definition can be derived when considering the received service and the caused interference to a task in a specific time window. An example, how this can be derived, is shown in Figure 4.2 representing a setup with four time partitions. The scheduling is fixed and repeated each T_{TDMA} time units. If we start on the left, P_1 is scheduled first. After T_1 time units P_1 is preempted for $T_{TDMA} - T_1$. Within the first T_{TDMA} time units the received service of each P_p is T_p and the received interference is $T_{TDMA} - T_p$. If we consider a time window of size T_{TDMA} and move it across the shown schedule, the received service as well as the received interference is always the same. We get therefore for any time window of size T_{TDMA} the following received service $\widehat{b_p}$ and received interference $\widehat{I_p}$:

$$\widehat{b_p}(\Delta t = T_{TDMA}) = T_p \tag{4.4}$$

$$\widehat{I_p}(\Delta t = T_{TDMA}) = \sum_{j \in \{\Gamma_{HYP} \setminus p\}} T_j = T_{TDMA} - T_p \tag{4.5}$$

Both values only depend on static parameters of the TDMA scheduling and not on the behavior of another partition. This leads us to the following generic definition of temporal isolation.

Definition 4.2: Temporal isolation

A scheduler $\mathcal{S}_i$ provides temporal isolation to a task τ_j on ϕ_i if

$$\widehat{I}_j^{\mathcal{S}_i}(\Delta t) = f_{\widehat{I}}(\Delta t, \tau_j, \mathcal{S}_i) \tag{4.6}$$

$$\widehat{b}_j^{\mathcal{S}_i}(\Delta t) = f_{\widehat{b}}(\Delta t, \tau_j, \mathcal{S}_i) \tag{4.7}$$

where both the received interference $\widehat{I}_j^{\mathcal{S}_i}(\Delta t)$ and the received service $\widehat{b}_j^{\mathcal{S}_i}(\Delta t)$ on ϕ_i only depend on the size of the considered time window Δt, the used scheduler $\mathcal{S}_i$ and the considered task τ_j. The behavior of any other task $\tau_k \in \{\Gamma_i \setminus \tau_j\}$ on ϕ_i does not have any effect on $\widehat{I}_j^{\mathcal{S}_i}(\Delta t)$ or $\widehat{b}_j^{\mathcal{S}_i}(\Delta t)$.

4.2 IRQ handling in virtualization environments

While the non work conserving scheduling with strict partitioning is already less than optimal for tasks, it gets even worse for IRQs which usually require particularly short response times. Access to the interrupt controller of the underlying hardware is only possible from the hypervisor. Otherwise, spatial as well as temporal isolation might be violated. As result, the POS does not directly interact with the physical interrupt controller but with a virtualized instead. How this virtutalized interrupt controller is implemented depends on the used soft- as well as hardware. As an example, [41] uses a software based IRQ virtualization by means of queues. Since hardware support for virtualization was integrated into modern architectures, the usage of a second virtualized register shows a hardware based integration [10]. Nevertheless, both methods behave similar when processing IRQs in a hypervisor.

Modern operating systems like Linux divide the IRQ processing in different parts, executing different tasks on different access layers. In case of Linux, the so called Top Half (TH) is executed directly in the ISR and is responsible for periphery dependent actions. This may include resetting IRQ flags as well as reconfiguring hardware registers. Often the TH is executed none-preemptive in a

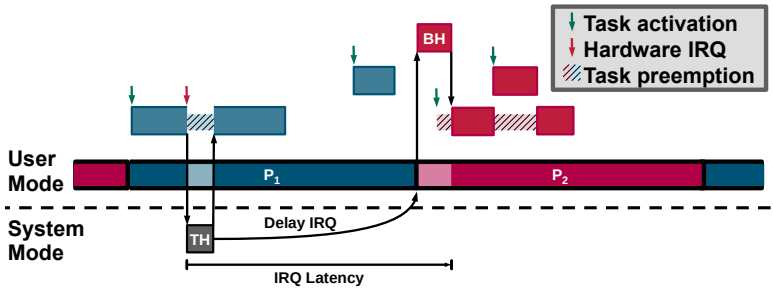


Figure 4.3: IRQ latency

critical section and therefore needs to very short. The real computation intensive work is then redirected to the Bottom Half (BH), which is not executed as a part of the ISR. This method of a distributed IRQ processing can also be used in a hypervisor. TH is then executed as part of the hypervisor and BH as part of the corresponding application. Usually the priority of a BH inside the partition is relatively high compared to the application. A BH is therefore often executed on the highest priority. An example for this behavior is shown in Figure 4.3. Let us assume an IRQ relevant for P_2 occurs while P_1 is scheduled by the hypervisor. The IRQ is recognized by the physical interrupt controller, the corresponding service routine in the hypervisor is entered and the TH of the associated interrupt source is executed. What happens next depends on the actual scheduled partition. If the IRQ belongs to the currently scheduled partition, it is forwarded directly to the POS and processed after the context has been switch back to user mode by the CPU. This is not the case in our example from Figure 4.3. As already mentioned, the IRQ is not relevant to the currently scheduled partition. Due to the temporal isolation, it isn't possible to execute the BH from P_2 while P_1 is scheduled. Therefore, the execution of the corresponding BH must be delayed until P_2 has been selected by the hypervisor for execution.

As shown in Figure 4.3 the IRQ latency between occurrence and finished processing is primarily dominated by the TDMA scheduling. If the IRQ occurs aligned with the TDMA schedule, it can be forwarded directly to the partition. However, for the worst-case we must assume that the corresponding partition just finished execution. As already mentioned, executing a BH from one partition in the context of another partition violates the temporal isolation. We already showed, that a partition doesn't necessary need its entire timeslot in order to finish all internal task executions in time. Therefore, the partition slack could be used to improve the IRQ processing. At that point we would violate the temporal isolation but as long as only the partition slack is used, no partition internal task

would miss its deadline. With other words this means, that the behavior of other tasks or IRQs may interfere with another task but only to a certain degree. This leads us to the definition of the sufficient temporal isolation bound.

Definition 4.3: Sufficient temporal isolation bound

A scheduler $\mathcal{S}_i$ provides a sufficient temporal isolation bound to a task τ_j on ϕ_i if

$$\tilde{I}_j^{\mathcal{S}_i}(\Delta t) = f_{\tilde{I}}(\Delta t, \Gamma_i, \mathcal{S}_i) \leq \widehat{I}_j^{\overline{\mathcal{S}}_i}(\Delta t) \tag{4.8}$$

$$\tilde{b}_j^{\mathcal{S}_i}(\Delta t) = f_{\tilde{b}}(\Delta t, \Gamma_i, \mathcal{S}_i) \geq \widehat{b}_j^{\overline{\mathcal{S}}_i}(\Delta t) \tag{4.9}$$

where both the received interference $\tilde{I}_j^{\mathcal{S}_i}(\Delta t)$ and the received service $\tilde{b}_j^{\mathcal{S}_i}(\Delta t)$ on ϕ_i only depend on the size of the considered time window Δt, the used scheduler $\mathcal{S}_i$ and the corresponding task-set Γ_i. The behavior of any other task $\tau_k \in \{\Gamma_i \setminus \tau_j\}$ on ϕ_i may affect $\tilde{I}_j^{\mathcal{S}_i}(\Delta t)$ or $\tilde{b}_j^{\mathcal{S}_i}(\Delta t)$, but the impact on both is limited to the behavior of a scheduler $\overline{\mathcal{S}}_i$ which would provide temporal isolation to the same task-set. Both, received interference and received service only need to satisfy (4.8) or (4.9) as long as τ_j is not idle.

Therefore, as long as the caused interference is never higher or the received service is never lower compared to temporal isolation for a task with outstanding workload, the sufficient temporal isolation bound is valid.

4.3 Monitoring based IRQ shaping in partitioned virtualization environments

The fast processing of IRQs is a well-known topic in standard OSs or small Real-Time OSs (RTOSs). Nevertheless, the latencies in virtualization environments have only received limited attention in the literature so far. Most of the work done so far addressed only latencies with the hypervisor, not including the BH in the application.

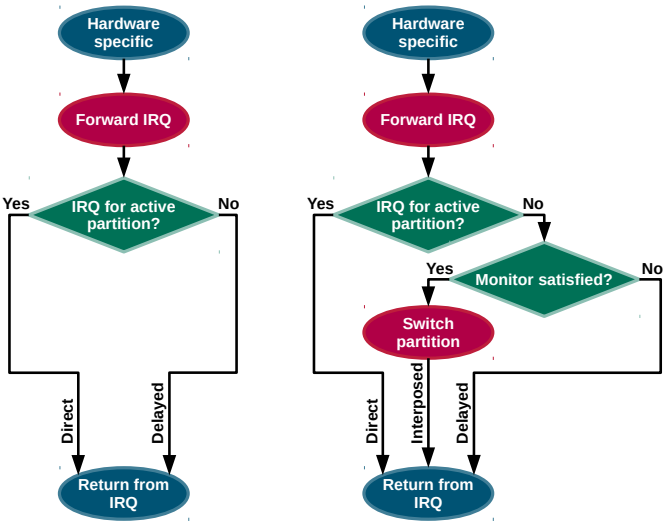
As an example, [91] examined credit-based scheduler of the Xen hypervisor with its shortcoming for IRQ latencies. The credit-based scheduler boosts partitions (called domains) to a higher scheduling priority if they receive an IRQ. While this shortens the response times for IRQs it doesn't ensure temporal isolation, as the partitions are immediately boosted even if it causes a deadline miss in another partition. Even the fact that the partitions were only boosted for one short time slice in order to respond on the IRQ, a number of IRQs large enough still violates temporal isolation. This approach was later refined in [72]

to account for finer granularities than the tick based accounting of the original credit-based system, but again without regarding temporal isolation. Besides other works [115] compares the interrupt latency observed by Xen, KVM, and a standard Linux. Again, it is shown that the lack of temporal partition enforcement within Xen is not suitable for real-time workloads.

Beside the work on general purpose virtualization environments like Xen, also the embedded domain was targeted. The work in [33] presented a method on shortening the interrupt latencies through manipulation of kernel operations within the formally verified seL4 kernel [57]. The idea of the paper is the reduction of non-preemptive sections in the kernel by inserting specific preemption points into those sections. While these reduces the IRQ latencies within the hypervisors kernel, it does not address the issue of IRQ latencies on POS level.

PikeOS [70] has the ability to implement IRQ handling in a special partition. Like in ARINC653 PikeOS provides a priority based scheduling inside each partition. All time partitions are then scheduled with TDMA. In addition to the that, the *time partition zero* is always active. On a scheduling decision, the PikeOS dispatcher chooses between the task with the highest priority in time partition zero and the task in the time partition within the current TDMA slot. In case of a greater or equal priority the task from time partition zero is executed. Time partition zero in PikeOs can be used for two different things. First, IRQ processing if the corresponding BH is executed on high priority inside partition zero. Or second, the consumption of idle times within the current TDMA slot in case of a lower priority. For a high priority in time partition zero, the current time partition of the TDMA scheduled is always preempted when executing tasks or BHs with a higher priority from time partition zero. But again, while these reduces the IRQ latencies drastically, using time partition zero for IRQ processing in PikeOS also violates the temporal isolation like the priority boost in Xen.

All proposed methods weaken the scheduling by treating IRQ handling on a different priority. But without monitoring such IRQs, neither temporal isolation nor sufficient temporal isolation can be provided as the interference isn't bounded. In [96] the authors tried to throttle the interference caused by IRQs directly in hardware at the source. They monitor the incoming IRQs and if a predefined limit has been reached, the IRQ source is blocked to the CPU until a new IRQ is permissible again. This method could be used to provide a sufficient temporal isolation as the activation of a BH inside a partition directly corresponds to the hardware IRQ. This way the caused interference would be bounded. Nevertheless, limiting IRQs at the source is not versatile enough as it doesn't work on shared hardware peripherals. As an example, let us assume a setup with two partitions $P_1 \& P_2$, where both h would like to react on incoming messages from the same interconnect. If the IRQ source of the corresponding interconnect get blocked in hardware due to a violation of sufficient temporal isolation from P_1,



(a) Without monitoring (b) With monitoring

Figure 4.4: basic IRQ handling inside the hypervisor

also all relevant IRQs for P_2 would be blocked.

In order to provide an improved IRQ handling we propose an IRQ shaping, based on the monitoring introduced by [87]. The general idea is to switch temporary the partition context in order to execute the BH and preserving a sufficient temporal isolation through the monitor. For a versatile behavior the monitoring is implemented as part of the IRQ handling inside the hypervisor. Figure 4.4 compares both the handling without (Figure 4.4a) and with (Figure 4.4b) monitoring. First of all, in both cases the hardware specific IRQ handling is executed. This might include reading registers as well as resetting flags. Second the IRQ is forwarded to the partition-level. How this is performed depends on the used hypervisor and POS. As an example, the evaluation setup from Chapter 8 with μC/OS-MMU as hypervisor and μC/OS-II as POS uses software queues for BH processing. Therefore, in case of μC/OS-MMU and μC/OS-II the IRQ would be pushed to a partition specific queue. Without monitoring, the hypervisor returns from the IRQ and executes the previously partition. Solely for evaluation purpose we label the IRQs according the occurrence over time. If an IRQ belongs to the previously preempted partition, it is labeled as *Direct*. If the corresponding partition isn't scheduled when the IRQ occurs, it is labeled as *Delayed*, which would be the case for the example from Figure 4.3.

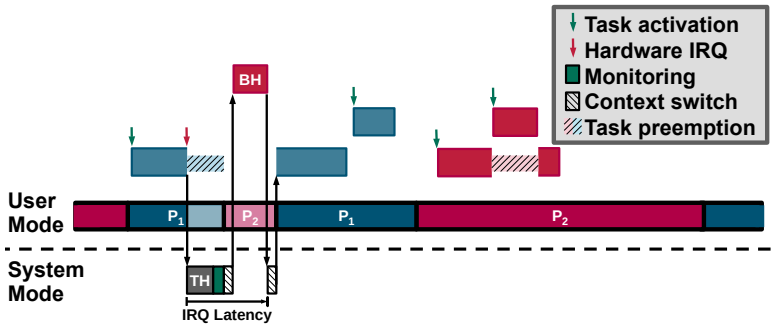


Figure 4.5: IRQ latency with shaping

Adding a monitoring as shown in Figure 4.4b only affects the handling for non-*Direct* IRQs. Instead of delaying the IRQ directly, a monitoring algorithm checks if an interposed execution of the BH would violate the sufficient temporal isolation of the preempted partition. If an execution is allowed, the stored partitions context is changed so that when the hypervisor returns from IRQ to partition context, the partition of the corresponding BH is executed. This additional case is labeled as *Interposed* in Figure 4.4b. The resulting behavior for the partition schedule from Figure 4.3 is presented in Figure 4.5. After handling the hardware specific stuff and forwarding the IRQ to P_2, the monitoring mechanism is entered. The monitoring mechanism from [87] uses minimum distance functions as bounds for incoming events (or IRQs) and performs with a constant runtime overhead. Next, the context is switched to P_2 and the corresponding BH is executed. In order to ensure a sufficient temporal isolation the execution of the interposed BH needs to be accounted, such that only the unused partition slack of P_1 (or any other partition than P_2) is used. Therefore, the hypervisor switches automatically to P_1 after a defined time (usually the WCET of the corresponding BH). Alternatively, P_2 should also be able to switch back on its own through a system call if the BH finishes earlier.

The amount of time that can be used to serve BHs from other partitions is limited to the partition slack, as this time is not necessary needed to finish all partition internal task activations in time. The monitoring from [87] can be implemented with a constant runtime and memory overhead. In general, it works on the same minimum distance functions as used for the multiple event busy-window. Therefore, $\delta^-(q)$ returns the minimum distance between the first and the $q'th$ consecutive event. Using this as a monitoring constraint means, that the last q events must be at least be separated by the according minimum distance. The authors in [87] proved, that for a certain group of distance functions

Listing 4.1: Simple minimum distance monitoring

```
#define Yes 1
#define No  0
unsigned int monitor_satisfied(mon_t *mon, unsigned int timestamp)
{
  unsigned int q = 2;
  unsigned int result = Yes;
  mon->add_to_tb(timestamp);           /* Add current event to tracebuffer (tb). */
  for (q = 2; q <= mon->l; q++)        /* Check if the last l events satisfy the */
  {                                    /* minimum distance function.             */
    if(d_in_tb(mon,q) < min_d(mon,q)){ /* Set "return" to NO if an event violates*/
      result = No;                     /* the minimum distance.                  */
    }
  }
  return result;
}
```

it is sufficient to only check the last l events, which limits the buffer needed for the monitoring. An example implementation for such a monitoring is shown in Listing 4.1. The function *monitor_satisfied()* gets a monitor handle (*mon*) and the *timestamp* of the current event (IRQ in our case) as parameter. Return value of the function is either 1 or 0, representing "Yes" or "No" from Figure 4.4b. First, the timestamp of the current event is added to the tracebuffer of the corresponding monitor. Next, for the last l events the temporal distance inside the tracebuffer is checked against the defined minimum distance function of the monitor. If for one of the checked distances the defined minimum distance is larger than the actual observed distance, the function returns "No". Otherwise, the function returns "Yes" and the BH can be interposed.

Both, runtime and memory overhead of Listing 4.1 depend directly on the trace length (*mon->l*). From the available partition slack $T_{p,S}$ and the worst case execution time of the interposed BH, we can derive the needed trace length as well as the minimum distance function. Let us assume that the BH itself has an WCET of $\overline{C}_{BH}$. Additionally, the needed time for context switching ($\overline{C}_{Ctx}$) and monitoring ($\overline{C}_{Mon}$) needs to be accounted. The allowed number n_p of interposed BHs according to the behavior from Figure 4.5 is then given as:

$$n_p = \left\lfloor \frac{T_{p,S}}{\overline{C}_{BH} + \overline{C}_{Mon} + 2 \cdot \overline{C}_{Ctx}} \right\rfloor \tag{4.10}$$

At this point it is important, that $\overline{C}_{Mon}$ itself depend on n_p as the loop in Listing 4.1 is executed n_p-times. This can be neglected if $\overline{C}_{BH} + 2 \cdot \overline{C}_{Ctx} >> \overline{C}_{Mon}$, otherwise we need to modify the previous equation. First we substitute with

$$\overline{C}_A = \overline{C}_{BH} + 2 \cdot \overline{C}_{Ctx}, \qquad \overline{C}_{Mon} = \overline{C}_B \cdot n_p$$

where $\overline{C}_B$ is the WCET of one monitoring loop iteration. With this we get

$$
\begin{aligned}
n_p &= \frac{T_{p,S}}{\overline{C}_A + n_p \cdot \overline{C}_B} \\
n_p \cdot (\overline{C}_A + n_p \cdot \overline{C}_B) &= T_{p,S} \\
{n_p}^2 + n_p \cdot \frac{\overline{C}_A}{\overline{C}B} - \frac{T_{p,S}}{\overline{C}B} &= 0
\end{aligned}
$$

which can be solved with the PQ-equation to:

$$
n_p = -\frac{\overline{C}_A}{2 \cdot \overline{C}_B} \pm \sqrt{\left[\frac{\overline{C}_A}{2 \cdot \overline{C}_B}\right]^2 + \frac{T_{p,S}}{\overline{C}B}}
$$

As all times are positive we can ignore the negative solution. Also, we need to apply the floor function as we can't handle half BHs. This leads us to

$$
n_p = \left\lfloor -\frac{\overline{C}_A}{2 \cdot \overline{C}_B} + \sqrt{\left[\frac{\overline{C}_A}{2 \cdot \overline{C}_B}\right]^2 + \frac{T_{p,S}}{\overline{C}B}} \right\rfloor \tag{4.11}
$$

representing a more accurate solution for the number of allowed BHs.

The monitoring must enforce that within a time cycle of T_{TDMA} length only n_p events are allowed. On the other hand this means that the $n_p + 1$ event needs to be at least T_{TDMA} time units separated. This leads to a monitoring length of $l_p = n_p + 1$. The corresponding minimum distance function $\delta^-_{BH,p}(q)$ is then given as:

$$
\delta^-_{BH,p}(q) = \begin{cases} 0, & q \leq n_p \\ T_{TDMA}, & q > n_p \end{cases} \qquad q \in [2 \ldots n_p + 1] \tag{4.12}
$$

With (4.10) and (4.12), a monitor can be configured in order to enforce sufficient temporal isolation. Nevertheless, a formal description for IRQ response times is still missing. Also, we still don't know how the partition slack, available for interposed BH processing, can be determined. For this reason, the following two sections will deal with exact those topics.

4.4 WCRT analysis

For the response time analysis, we will provide different definitions for the blocking term $B_{p,i}(\Delta t)$ from (4.3). The different blocking terms will be named according to their scheduling techniques. With $B^{TSPP}_{p,i}(\Delta t)$ we provide a first block-

ing term for a partition-level TDMA $B_p^{TDMA}(\Delta t)$ and task-level SPP $B_{p,i}^{SPP}(\Delta t)$ scheduling.

$$B_{p,i}^{TSPP}(\Delta t) = B_p^{TDMA}(\Delta t) + B_{p,i}^{SPP}(\Delta t) \tag{4.13}$$

In order to include also the interference caused by IRQs, we need an additional $B_{IRQ}(\Delta t)$. This leads to a $B_{p,i}(\Delta t)$ given as

$$B_{p,i}(\Delta t) = B_{IRQ}(\Delta t) + B_{p,i}^{TSPP}(\Delta t) \tag{4.14}$$

where $B_{IRQ}(\Delta t)$ includes the blocking caused by each TH. In general, an IRQ can be described as a task on a high priority. $B_{IRQ}(\Delta t)$ includes the interference of all THs executed in system (or even hypervisor) mode of the underlying CPU. Therefore, $B_{IRQ}(\Delta t)$ can be written as

$$B_{IRQ}(\Delta t) \sum_{j \in \Gamma_{IRQ}} \eta_j^+(\Delta t) \cdot \overline{C}_j \tag{4.15}$$

where Γ_{IRQ} includes all available IRQ sources and $\overline{C}_j$ describes the worst-case execution time of the corresponding TH. The activation pattern is abstracted via η_j^+. Possible activation pattern in the automotive domain for IRQs range from a sporadic or periodic bursts to a purely sporadic or periodic behavior.

As already discussed, the IRQ handling is divided into two parts. We already characterized the TH as a task within the context of the hypervisor, same can be done within the partition context for each BH. Therefore, the different BHs inside a partition can be described as a simple task with an execution time $\overline{C}_{p,i}$ and a priority higher than the application related task. The activation pattern of a BH is the same as the corresponding TH executed in the hypervisor context. This leads us to

$$B_{p,i}^{SPP}(\Delta t) = \sum_{j \in hp(\tau_{p,i})} \eta_{p,j}^+(\Delta t) \cdot \overline{C}_{p,j} \tag{4.16}$$

where $hp(\tau_{p,i})$ is the set of tasks inside partition p with a higher or equal priority as the considered task $\tau_{p,i}$. This includes also all BHs inside the considered partitions.

As mentioned in Section 3.2 the busy-window based RTA was initially designed for work-conserving schedulers. TDMA at this point is not work-conserving as shown in Section 4.1 and Figure 4.1. A simple way to deal with this is, to describe all other time partitions as interfering tasks. The execution time of the interfering time partition is equal to their timeslot sizes. This leads to

$$B_p^{TDMA}(\Delta t) = \sum_{j \in (\Gamma_{HYP} \setminus p)} \eta_j^+(\Delta t) \cdot T_j$$

where $\eta_j^+(\Delta t)$ describes a periodic activation every T_{TDMA} time units for each time partition. Replacing the activation pattern with the actual periodic activation gives us

$$B_p^{TDMA}(\Delta t) = \sum_{j \in (\Gamma_{HYP} \setminus p)} \left\lceil \frac{\Delta t}{T_{TDMA}} \right\rceil \cdot T_j$$

where $\lceil \Delta t / T_{TDMA} \rceil$ could also be placed outside the sum. Considering (4.1) we can also rewrite the sum. As a result, the TDMA based blocking $B_p^{TDMA}(\Delta t)$ can be constructed only based on the corresponding slot size T_p and the overall TDMA cycle length T_{TDMA}.

$$B_p^{TDMA}(\Delta t) = (T_{TDMA} - T_p) \cdot \left\lceil \frac{\Delta t}{T_{TDMA}} \right\rceil \tag{4.17}$$

When considering response times for IRQs, a RTA of the corresponding BH provides the desired result. Due to the additional interference from all THs in $B_{IRQ}(\Delta t)$, the time needed for execution of the corresponding TH is directly in the considered interference. This is possible, as the execution of both, TH and BH can directly be used for the corresponding interference or task execution time. With the introduction of the monitoring based IRQ shaping, we need to modify this values, as additional software overhead for monitoring and context switching is introduced. Figure 4.4b and Figure 4.5 show the difference of executing an interposed or direct IRQ. In case of a direct IRQ, no monitoring inside TH and also no context switch for the corresponding BH is needed. Compared to Figure 4.4a, an implementation of Figure 4.4b must check the active partition, as the handling for a foreign partition is different. For a direct handled IRQ only an additional if-statement would be checked which could be neglected execution time wise. Nevertheless, the worst-case interference by IRQ handling is generated for an interposed IRQ. As a result we need to change the WCET values for both, TH and BH. The primary overhead introduced to the TH is due to the monitoring. Therefore, we get a modified TH WCET for IRQ source j, which can be used in (4.15).

$$\overline{C}_j = \overline{C}_{j,TH} + \overline{C}_{Mon} \tag{4.18}$$

In case of an interposed BH, the additional overhead is based on the introduced context switches. Finally, we get a modified BH WCET for IRQ source j inside partition p, which can be used in (4.16).

$$\overline{C}_{p,j} = \overline{C}_{p,j,BH} + 2 \cdot \overline{C}_{Ctx} \tag{4.19}$$

With (4.18) and (4.19) the response times can be calculated based on (4.14). Under consideration that all IRQs adhere to the monitoring and the available slack, $B_p^{TDMA}(\Delta t)$ could be removed from $B_{p,i}^{TSPP}(\Delta t)$, leading to the desired behavior from Figure 4.5.

4.5 Time Cycle Optimization

The described IRQ shaping from Section 4.3 shows how to improve IRQ response times due to the use of partition slack. Nevertheless, Section 4.3 does not consider the design of a partition setup which can benefit from this method. Within the context of ARINC653, several approaches for time partition calculation are available. One of the most frequently mentioned works within this context is [76]. The authors of [76] provided an overview of the APEX interface and introduced a method to optimize the time partitions with RMS inside. This method can now be used for the described analysis in Section 4.4, as IRQs can rarely be described by RMS. Another optimization method for time partitions was provided in [111]. They used a meta-heuristic approach called *simulated annealing* to optimize time partitions. Nevertheless, the used system model does not consider a scheduling on partition-level. To solve this issue, we show in this section how a partition setup with maximized partition slack can be generated. As the optimization algorithm is based on the analysis introduced in Chapter 3 and extended in section 4.1/4.4, it is able to handle arbitrary IRQ/task activation pattern as well as arbitrary deadlines.

4.5.1 Optimization algorithm

Figure 4.6 shows the proposed optimization algorithm. General idea of the algorithm is to calculate a minimum timeslot size $T_{p,min}$ for each of the Ω partitions and a given T_{TDMA}. From those values the slack in the entire TDMA cycle can be derived and distributed to the different partitions. For the optimization algorithm it is necessary to calculate an upper bound for the TDMA cycle length T_{TDMA}. The upper bound $\widehat{T}_{TDMA}$ is used as starting condition for the optimization algorithm. We will first show the concept of the algorithm and provide a formal proof for the starting condition afterwards in Section 4.5.2.

The algorithm starts on the upper left corner with assigning the upper bound $\widehat{T}_{TDMA}$ to T_{TDMA}. Next the partition analysis is started, which returns a tuple of minimum partition sizes $(T_{1,min}, \ldots, T_{\Omega,min})$ for the current T_{TDMA}. The calculation of the $T_{p,min}$ values is shown on the right side of Figure 4.6 and will be

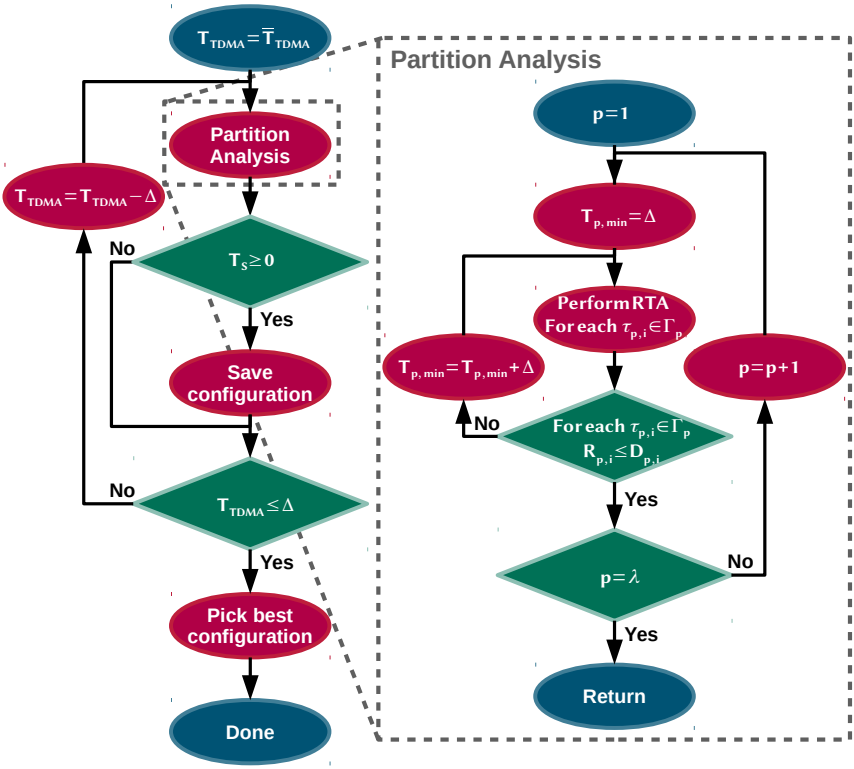


Figure 4.6: Laxity based upper bound for T_{TDMA}

explained later. The partition analysis calculates for each partition the smallest time partition, so that it reaches all deadlines at the considered T_{TDMA}. As we know from (4.1), the sum of all time partitions must be T_{TDMA}. At this point it is possible that the sum over all calculated minimum time partition sizes is smaller, equal or larger than T_{TDMA}. Therefore, we introduce the TDMA cycle slack T_S as difference between considered T_{TDMA} and minimum time partition sizes.

$$T_S = T_{TDMA} - \sum_{p=1}^{\Omega} T_{p,min} \tag{4.20}$$

According to the sum over the minimum time partition sizes, T_S can be smaller, equal or larger than zero:

$T_S < 0$: If the time cylce slack is negative, the sum of the required minimum time partition sizes is larger than T_{TDMA}. This means, that the time partitions need more execution time than actual available. The partition setup is not schedulable for the considered T_{TDMA}.

$T_S = 0$: If the time cylce slack is zero, the needed minimum execution time by the time partitions is equal to the available time during a TDMA cycle of size T_{TDMA}. This means, that the partition setup is schedulable for the considered T_{TDMA}, but it does not provide any slack for additional IRQ handling.

$T_S > 0$: If the time cylce slack is positive, the needed minimum execution time by the time partitions is smaller than the available time during a TDMA cycle of size T_{TDMA}. This means, that the partition setup is schedulable for the considered T_{TDMA} and the available slack can be used for additional IRQ handling.

The target of the optimization algorithm is to find all possible time cycle configurations which provide a schedulable system. Therefore, only time cycle configurations with $T_S \geq 0$ are saved for later use in Figure 4.6.

Next, the current T_{TDMA} is checked against a step size Δ. If $T_{TDMA} \leq \Delta$, the calculation of possible time cycle configurations is stopped. Otherwise, the T_{TDMA} is decreased by Δ and the partition analysis is restarted for the new T_{TDMA}. The granularity of Δ directly influences the optimization algorithm. A smaller Δ increases both, runtime and accuracy of the optimization algorithm. While a bound on the granularity is usually independent of the executed software, the underlying hardware instead provides a lower bound for Δ, as the TDMA schedule needs to be generated or controlled by a hardware timer. The Δ can therefore directly be derived by the granularity of the hardware timer used of schedule generation. Choosing a finer granularity than the hardware can provide only increases the algorithm runtime, but does not improve the accuracy. After the partition analysis has been performed for the last T_{TDMA}, the best solution is picked and the available time cycle slack is distributed to the partitions. A better explanation of this process will be given later in Section 4.5.3.

After knowing the outer layer of the optimization algorithm, we can now focus on the partition analysis on the right of Figure 4.6 which calculates the needed minimum time partition sizes for a given T_{TDMA}. The analysis starts with the first partition and assigns Δ as minimum time partition size $T_{p,min}$. Next, a RTA based on (4.14) is performed. Afterwards the schedulability is checked according to Definition 3.16. If one of the task deadlines in the corresponding partition is missed, $T_{p,min}$ is increased by Δ and the RTA is repeated. This is done until all tasks inside the considered partition reach their deadlines and repeated for all Ω

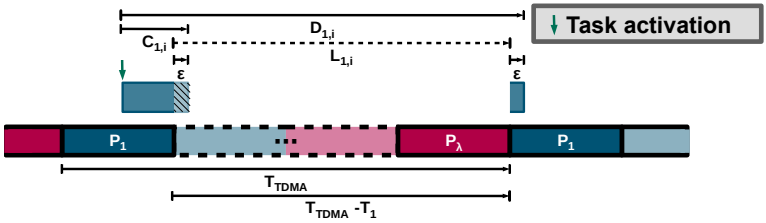


Figure 4.7: Laxity based upper bound for T_{TDMA}

partitions. Due the fact, that (4.14) only depends on partition internal parameters and the given T_{TDMA}, the inner loop of the partition analysis can be executed in parallel. As a result the partition analysis returns a tuple of minimum time partition sizes which represent the minimum time each partition needs for the given T_{TDMA}.

4.5.2 Laxity based time cycle bound

The optimization algorithm simply searches for possible time cycle configurations in a given interval. While the lower bound is provided by the step size granularity, the upper bound is given as $\widehat{T}_{TDMA}$. The information included in $\widehat{T}_{TDMA}$ is that the considered system is impossible to schedule for a cycle length $T_{TDMA} > \widehat{T}_{TDMA}$. Therefore, $\widehat{T}_{TDMA}$ provides an upper bound, which can be used by the algorithm in Figure 4.6.

Lemma 4.1: Upper bound for TDMA cycle length

A sufficient upper bound $\widehat{T}_{TDMA}$, without considering additional task activations for higher prior tasks within the same partition is given through:

$$\widehat{T}_{TDMA} = \frac{\sum_{p=1}^{\Omega} \min_{\tau_{p,i} \in \Gamma_p} \{D_{p,i} - \overline{C}_{p,i}\}}{\Omega - 1} \tag{4.21}$$

The proof of this theorem can be derived based on Figure 4.7. Let us assume that a task $\tau_{1,i}$ in P_1 is activated such that it is impossible to finish the execution during the current TDMA cycle. Resulting from this, a small amount time (ϵ) is needed to finish the execution within the following cycle as shown in Figure 4.7. For the execution of $\tau_{1,i}$ we assume that no interference is caused by other tasks from P_1. As a result we can see in Figure 4.7, that the maximum amount of time

$\tau_{1,i}$ can be preempted by the TDMA schedule without missing its deadline is limited to the laxity of the considered task.

$$L_{1,i} = D_{1,i} - \overline{C}_{1,i} \geq T_{TDMA} - T_1 \tag{4.22}$$

If we would consider the activation patterns and include partition internal blocking, the maximum TDMA cycle length would be decrease as the maximum allowed interference is still limited by the laxity. While (4.22) is only valid for the considered task in the example from Figure 4.7, we need to generalize this approach for an entire task-set inside a partition. A sufficient bound is provided if we use the task with the minimum laxity inside a partition p. This leads us to

$$L_p = \min_{\tau_{p,i} \in \Gamma_p} \{D_{p,i} - \overline{C}_{p,i}\} \tag{4.23}$$

where we consider L_p as the minimum *partition laxity* of partition p. Combining (4.22) and (4.23) leads us to.

$$L_p = \min_{\tau_{p,i} \in \Gamma_p} \{D_{p,i} - \overline{C}_{p,i}\} \geq T_{TDMA} - T_p = \sum_{i=1}^{\Omega} T_i - T_p \tag{4.24}$$

When considering the maximum allowed blocking time, the minimum partition laxity is equal to the interference caused by a TDMA cycle with length $\widehat{T}_{TDMA}$.

$$L_p = \widehat{T}_{TDMA} - T_p = \sum_{i=1}^{\Omega} T_i - T_p \tag{4.25}$$

If we evaluate this for all Ω partitions, we can describe it as a system of linear equations for better illustration.

$$\underline{A}_{\Omega,\Omega} = \begin{bmatrix} 0 & 1 & \dots & 1 & 1 \\ 1 & 0 & \dots & 1 & 1 \\ \vdots & & & & \vdots \\ 1 & 1 & \dots & 1 & 0 \end{bmatrix}, \vec{t} = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_\Omega \end{bmatrix}, \vec{l} = \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_\Omega \end{bmatrix}$$

$$\underline{A}_{\Omega,\Omega} \cdot \vec{t} = \vec{l}$$

The multiplication $\underline{A}_{\Omega,\Omega} \cdot \vec{t}$ gives us:

$$\begin{bmatrix} 0 + T_2 + \dots + T_{\Omega-1} + T_\Omega \\ T_1 + 0 + \dots + T_{\Omega-1} + T_\Omega \\ \vdots \\ T_1 + T_2 + \dots + T_{\Omega-1} + 0 \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \\ \vdots \\ L_\Omega \end{bmatrix} \tag{4.26}$$

Now we are close to finish. If we take a closer look on the left side of the equation (4.26), we can see that in each row one time partition is missing. As an example, due to (4.23) T_1 is missing for L_1, T_2 is missing for L_2 and so on. This means, that each time partition is included $(\Omega - 1)$ times in the listed Ω rows. Calculating the sum over the Ω rows leads to:

$$(\Omega - 1) \cdot (T_1 + T_2 + \ldots + T_\Omega) = \sum_{p=1}^{\Omega} L_p$$

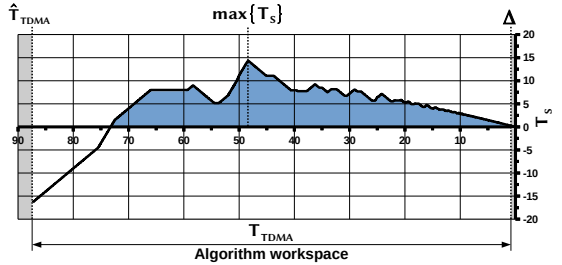
Replacing the sum of time partitions with $\widehat{T}_{TDMA}$ and L_p with (4.23) finalizes the proof of Lemma 4.1.

$$\begin{aligned} (\Omega - 1) \cdot \widehat{T}_{TDMA} &= \sum_{p=1}^{\Omega} \min_{\tau_{p,i} \in \Gamma_p} \{D_{p,i} - \overline{C}_{p,i}\} \\ \widehat{T}_{TDMA} &= \frac{\sum_{p=1}^{\Omega} \min_{\tau_{p,i} \in \Gamma_p} \{D_{p,i} - \overline{C}_{p,i}\}}{(\Omega - 1)} \end{aligned} \tag{4.27}$$

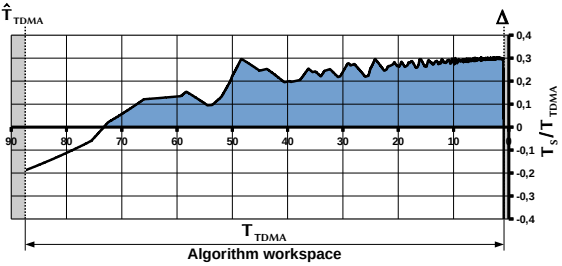
4.5.3 Slack distribution

The previous two sections explained how the time cycle configurations with additional slack can be calculated. Nevertheless, we neither explained in Section 4.5.1 how to pick one of the possible solutions nor how the calculated slack is assigned to different partitions. The reason for this is that the strategies for both tasks may depend on the actual implemented applications. Figure 4.8 shows the available slack over all checked TDMA cycles. The shown values in Figure 4.8 have been generated for a partition setup which will be explained and used later during evaluation in Section 8.1.2. All times labeled on the axis are given in milliseconds.

Figure 4.8a shows the absolute time cycle slack over the entire algorithm workspace from $\widehat{T}_{TDMA}$ down to Δ. Which of the possible configurations is "the best solution" can't be said directly. As we can see, a global maximum for T_S is given for a setup with $\sim 48ms$. At first glance one might think, that the absolute maximum slack always provides the best partition configuration, but this doesn't need to be the case. The time cycle slack is available each T_{TDMA} time units, therefore it is possible that a configuration with less slack during a time cycle provides more slack within a larger time window. In order to visualize this, Figure 4.8b shows the available slack normalized to the corresponding T_{TDMA}. By coincidence for the used example, the global maximum at $\sim 48ms$ is still a maxi-



(a) Absolute



(b) Normalized

Figure 4.8: Available slack within algorithm workspace

mum even if it is normalized to the corresponding time cycle. Nevertheless, several other configurations provide the same amount of normalized slack. At this point it depends on the actual applications inside, which of the possible configurations fits the needs best. A longer time cycle reduce the number of context switches, which is often a desirable goal. Especially when using a method like the described IRQ shaping which introduces additionally context switches. Also, a longer time cycle and therefore longer partitions may also shorten the response times of the internal tasks as those may finish within one time cycle. On the other hand, if a task doesn't finish its execution within one time partition, the delay caused by the TDMA scheduling is large. Once a configuration has been picked for usage, the available time cycle slack needs to be distributed to the different time partitions. The distribution of the time cycle slack to the different partitions also heavily depends on the actual applications inside the partitions. If one partition implements more IRQ handling on partition-level it benefits most, if the slack is primarily distributed to the other partitions. Also, if the IRQs follow a specific event model (like periodic bursts), the slack distribution across the partition could be aligned to this event model. If none of the previous preconditions

are given, an equal slack distribution to the partitions could be desirable.

$$T_{p,S} = T_S \cdot \left\lfloor \frac{T_{p,min}}{\sum_{i=1}^{\Omega} T_{i,min}} \right\rfloor \tag{4.28}$$

In this case the time cycle slack is distributed to the different partitions based on their minimum time partition sizes. Based on Definition 4.1 and (4.28) the final time partition sizes can be calculated. As described in Section 4.3, the monitoring is then configured according to the different $T_{p,S}$ values.

Both, the selection of the "best" configuration and the "best" slack distribution heavily depend on the actual applications inside the partitions. As discussed an optimization for both depend on more than one factor, therefore we do not pinpoint one method as "the best solution". Without any knowledge about applications and/or IRQ pattern an equal distribution of the available slack based on a configuration with maximum slack per time cycle is a step towards the right direction. If several configurations provide the same amount of slack per time cycle, the configuration for the largest T_{TDMA} is desirable in order to minimize context switching.

4.6 Formal limitations of IRQ shaping

The monitoring based IRQ shaping provides a simple relaxation of the strict TDMA scheduling. When considering Section 4.3 and Section 4.4, we can clearly see that the IRQ processing is improved drastically for IRQs handled through the monitoring based shaping. This way the average response time for IRQs is reduced and the scheduling is improved, as certain idle times during scheduling can be used for IRQ processing. Nevertheless, the proposed method has certain disadvantages.

First of all, the IRQ shaping only improves the WCRT of IRQs, if all possible IRQs always occur during the time partition of the correct partition or when slack time is available. If this is not given due to insufficient slack or unknown behavior of IRQs, only the average response time is improved but not the WCRT. Second, the monitoring from [87] is not ideal for such an enforcement of temporal isolation as it does not scale ideally. As shown in Listing 4.1 both memory usage for trace buffer and the runtime overhead scale linear with number of events need to be checked. The calculation of the number of allowed events (or in our case IRQs) is given through (4.10) (or (4.11)). This shows directly that the monitoring for a system with a lot of slack generates more overhead than for a system with less slack. While more slack is available in theory, it can't be used as efficient as possible due to additional overhead. Third, even though we consider IRQs as tasks during the analysis, we still limit the usage of slack to IRQs. Indirectly

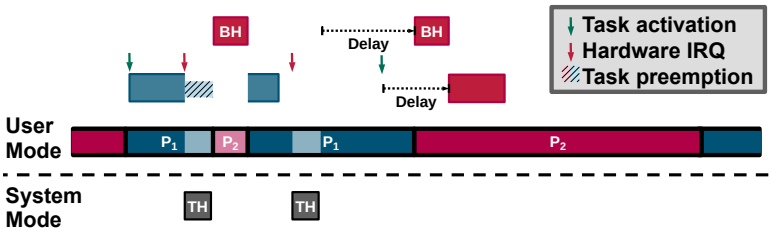


Figure 4.9: Limitation of IRQ shaping

this means, that a normal task can not use the available slack when it's activation occurs within a foreign time partitions. This leads us the fourth and worst disadvantage. The scheduling is still not work conserving.

Even for IRQs this is not the case as shown in Figure 4.9. Let us assume that the slack of P_1 allows the execution of one foreign BH within its time partition. As discussed, the slack calculation is based on the busy-window method which takes the critical instant as starting condition. It is therefore possible, that this worst-case behavior of simultaneous task activations is not always the case. The critical instant might include another task in P_1 which needs to be finished but isn't executed in each time cycle. As a result P_1 would be idle during this and as the calculated slack is based on the critical instant, no BH could be interposed. This is shown in Figure 4.9. First a BH is interposed during P_1. As a result the available slack $T_{1,S}$ is consumed and the currently executed task in P_1 is preempted. Second, the preempted task resumes and finishes its execution. At this point P_1 is idle, but since the time partition slack was already consumed, the following BH or task activations for P_2 must be delayed in order to satisfy the sufficient temporal isolation. Reason for this behavior is the fact, that the monitoring based shaping does not consider if the currently executing partition which should be preempted is idle or not. As a result, the monitoring doesn't know that interposing the BH a second time would not cause any deadline misses. In order to enable such a behavior we show in the following chapter how to modify an SPS based budget scheduling. Main objective is to provide a replacement of the previously described TDMA based scheduling, which takes account of each partitions actual required workload.

"I'm sciencing as fast as I can!"

- Professor Hubert J. Farnsworth

CHAPTER 5

Sporadic Server based Budget Scheduling

As discussed in Chapter 4, the TDMA based scheduling is less optimal for systems with sporadic events like IRQs. In order to react on such events sufficiently a more flexible scheduling, which still preserves a sufficient temporal isolation, is needed. This chapter shows how the existing SPS mechanism can be modified in order to replace a TDMA based scheduling as used in ARINC653. Most of the work used in this chapter was previously presented in [26] and [24].

The SPS was first proposed in [108] and has become a part of POSIX [113] as a scheduling policy. The general idea was to provide a performant scheduling mechanism with temporal isolation for aperiodic (sporadic) task activation. For each aperiodic task, the SPS defines an execution budget and a replenishment time. In contrast to other mechanisms like the deferrable server mentioned in [108] or the priority exchange server, the SPS does only replenish an execution budget if it was used before.

The basic functionality of the SPS mechanisms can be explained best with an example, like shown in Figure 5.1. Each task τ_i has an execution time budget b_i, which is initialized to its maximum value $b_{i,max}$ at the beginning. When a task τ_i is executed, the corresponding budget b_i is decreased. A downwards slope in Figure 5.1 therefore indicates a time range, where the corresponding task is picked by the scheduler for execution. Due to the specification of the SPS

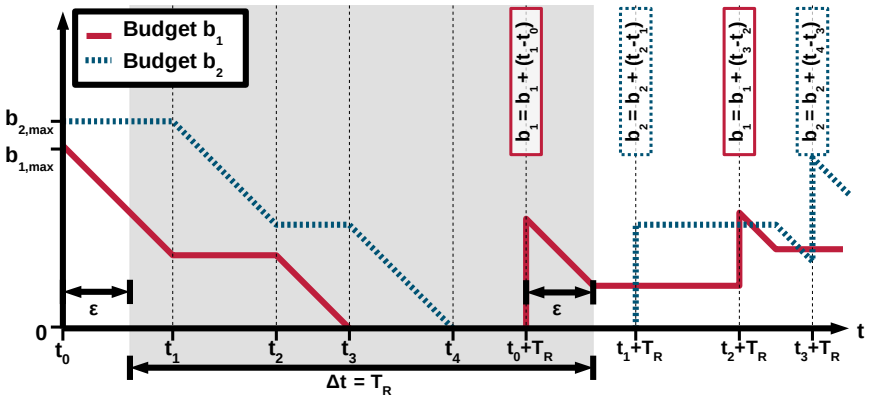


Figure 5.1: Laxity based upper bound for T_{TDMA}

algorithm from [108], each task τ_i has a replenishment period $T_{R,i}$. This way, used budget is always replenished relative to the start of the consumption (e.g. execution). In Figure 5.1 τ_1 starts execution at t_0 and stops at t_1. During this time a budget of $t_1 - t_0$ time units was consumed. At this point one obvious difference to a simple TDMA scheduling can be recognized. An SPS enforced task doesn't have to consume the available budget in one piece. If all outstanding workload has been processed, the task can self suspend its execution. As a result, nothing of the available budget is waisted.

For the sake of simplicity, both tasks in Figure 5.1 have the same replenishment time $T_{R,1} = T_{R,2} = T_R$. Due to this constraint, an amount of $t_1 - t_0$ time units is replenished at $t_0 + T_R$ and the available budget for τ_1 is increased. The same is done for τ_2 when it stops execution at t_2 and so on. [108] also defines priorities for sporadic server tasks. Those priorities specify the order in which the tasks are scheduled. In our example from Figure 5.1, τ_1 would have a higher priority as it is scheduled first.

The later defined POSIX standard states that each sporadic server task would have two priorities. One foreground priority, used as long as the task has remaining budget and a lower background priority, used when the budget is depleted. However, [109] already showed the pitfalls when it comes to the implementation of background scheduling. The first SPS specification in the POSIX standard was not entirely correct, causing *budget amplification*. The primary problem was an influenced budget replenishment based the execution during background scheduling. This caused an unreliable temporal isolation, since budget was replenished too early. Also, there have been suggestions to implement this policy into the Linux kernel [45] to provide additional real-time capabilities and tem-

poral isolation. The usage of an SPS for virtualization environments has already been proposed in [79, 36]. The described kernel Quest V uses so called VCPUs which are scheduled on different physical CPUs. Quest V differentiates between two types of VCPUs, called *Main VCPUs* for applications and *I/O VCPU* for IRQ based events. While the SPS was only considered for *Main VCPUs*, it was not used for the IRQ based workload of *I/O VCPUs*. Also, the authors in [117] used a standard priority based sporadic server to provide soft real-time guarantees. Nevertheless, the proposed method was not designed to cover hierarchical systems like ARINC653.

An analysis for tasks scheduled under a SPS policy has been proposed in [102, 8]. Nevertheless, the provided analysis did neither cover background scheduling nor is it applicable for a hierarchical scheduling with SPP as defined in ARINC653. In general, there was not a lot of work considering the SPS mechanism in the last years. A major reason for this is the tendency towards a complex implementation of an SPS based system with several servers and different replenishment periods. The benefit compared to other server based scheduling solutions might be moderate [37]. This chapter will show, that under certain constraints a modified SPS can be a valuable replacement of the strict TDMA based partition-level scheduling in ARINC653. The proof, that such a system can be implemented with a low overhead profile is later given in Chapter 7 and 8.

5.1 Isolation bound

One of the two major advantages of the SPS is its ability to provide a sufficient temporal isolation bound as required in Definition 4.3. If we consider a time window $\Delta t = T_R$ and move it across the time line, any task τ_i can never execute more than $b_{i,max}$ time units during this time window. The worst-case behavior of a task τ_i is given, when τ_i uses its budget as soon as it is available. In order to explain this further we use again Figure 5.1.

Let's have a look at τ_1 depleting its budget b_1 at t_3, which lays within the first T_R time units ($t_0 \rightarrow t_0 + T_R$). The caused interference by τ_1 within this time window is equal to its maximum value $b_{1,max}$ as the entire budget was used for execution. Right at $t_0 + T_R$ the budget of τ_1 is replenished and τ_1 starts executing again. If we shift the considered time window by ϵ time units to the right, the budget used by τ_1 inside the time window is still $b_{1,max}$ and can never be greater than that. The additional interference of size ϵ between $t_0 + T_R \rightarrow t_0 + T_R + \epsilon$ is equal to the interference which fell out of the considered window on the left. Therefore, the budget used by τ_1 inside the considered window is still the same

As already mentioned we consider the SPS under certain constraints as a replacement for the TDMA based partition-level scheduling in ARINC653. Based

on the SPS mechanism, within any considered time window of size $T_{R,p}$ a task τ_p, as part of the SPSs scheduled task-set Γ_{HYP}, uses never more budget than defined in $b_{p,max}$. If all SPS tasks use the same replenishment period T_R as shown in Figure 5.1, a single task will never see more interference than the summarized budgets of all other tasks in Γ_{HYP} within a time window $\Delta t = T_R$. This leads to a worst-case interference given as:

$$\bar{I}_p^{SPS}(\Delta t = T_R) = \sum_{j \in \{\Gamma_{HYP} \setminus p\}} b_{j,max} \tag{5.1}$$

When comparing this to the TDMA interference term from (4.5) one might directly recognize that the SPS mechanism provides the same isolation bound than TDMA if we interpret the time cycle length as replenishment period and timeslot sizes as maximum budgets.

$$p \in \Omega \; (T_{R,p} = T_{TDMA}, b_{p,max} = T_p) \tag{5.2}$$

Important at this point is that the SPS provides an interference bound in the terms of a sufficient temporal isolation as defined in Definition 4.3 and not a temporal isolation as defined in Definition 4.2. Reason for this is the possible self suspend of a task scheduled by the SPS. In case of TDMA the hypervisor would not leave the context of a partition if this partition is idle. Therefore, the caused interference in case of TDMA is in any time window of size $\Delta t = T_{TDMA}$ equal to the worst-case, even if a partition is idle. In case of the SPS mechanism this is not the case anymore as a partition can self-suspend its execution when it would be idle otherwise. The actual interference during $\Delta t = T_R$ therefore depends on the behavior of all other tasks/partitions with a TDMA-like limitation given by (5.1).

5.1.1 Scheduler setup

Due to the previous argumentation, the SPS bounds the interference according to Definition 4.3. But in order to provide a sufficient temporal isolation bound, also the service (execution budget) provisioning must be considered. Since we want to replace the scheduling algorithm of ARINC653, TDMA must be considered as a lower bound for service provisioning. In general, the SPS on its own does not decide which task or partition should be dispatched next by the scheduler. This wasn't even the case for the original POSIX definition. Instead, only the budget consumption and therefore the caused interference to other tasks is enforced. The POSIX definition of the SPS proposes a priority based scheduling, where tasks use two different priorities. One higher foreground priority used for a task τ_i while $b_i > 0$ and a lower background priority used while $b = 0$. A task/partition

is then scheduled according to the currently set priority whenever outstanding workload exists. Even though the SPS controls the priorities in case of the POSIX definition, it doesn't take an actual scheduling decision. As a flexible foundation we use a generic architecture, which divides the setup into two major parts. The actual implementation of the proposed architecture will be discussed later in Section 7.1 (Figure 7.2). This section only provides a coarse grained architecture overview, for better understanding of the theoretical considerations.

First, there is the standard SPS mechanism which enforces the interference. And second, there is the scheduler which is in charge of determining which partition should be scheduled next. The main duty of the SPS in order to provide an interference bound is the control of a timer module. The timer provides two different signals, one for budget depletion (*Empty*) and another one for budget replenishment (*Refill*). The *Empty* signal always belongs to the current scheduled partition, the *Refill* signal may belong to any partition on the same resource. In the proposed architecture, the SPS is also in charge of dispatching (*Dispatch*) partitions. This means, that the SPS is able to initiate a context switch if needed. The SPS also tracks, if a partition would like to suspend its execution (*Idle*). This happens when all tasks inside a partition served their current activation and do not have any outstanding workload. Allowing a partition to stop its execution is the second major advantage of the SPS, as it provides the ability to reduce unnecessary blocking times. Combining this with the optimization algorithm from Section 4.5, is the logical next step. The output of the optimization algorithm from Section 4.5 maximizes the slack inside the partitions and should therefore provide a maximized benefit from this *Idle*-feature. An idle partition can be reactivated with the *Resume* signal. In general, the *Resume* signal is issued when a task inside a partition is activated. Such an activation can be triggered by different events like time ticks, partition-2-partition communication or any other type of subscribed hardware IRQs.

We already mentioned, that the SPS itself doesn't decide which partition should be dispatched next. Instead, this decision is taken by the scheduler. General idea of the proposed architecture is, that the SPS forwards scheduling dependent events to the scheduler. Those events trigger callback functions inside the scheduler. The collection of events is described by the Callback API (CB API) which is explained later in Section 7.1.2 Each scheduler callback performs a scheduling decision and returns the partition which should be dispatched next by the SPS. If the returned partition differs from the currently executed one, a context switch is initiated. In addition to the callbacks, the scheduler also haves access to the current budget state of each partition. Which additional environment values are included for decision taking, depend on the implemented scheduling mechanism. This way the entire behavior of the system and the provided service for each partition is under control of the scheduler.

5.1.2 Service provisioning

In Section 4.1 we already mentioned the service provisioning for TDMA. In each time window $\Delta t = T_{TDMA}$ the received service of a partition is equal to the corresponding timeslot size T_p. If we transfer this to the SPS based scheduling this means, that a partition must always receive its maximum budget $b_{p,max}$ within a time window of size T_R. This leads us to Definition 5.1.

Definition 5.1: TDMA-like service provisioning

In order to provide a TDMA-like service, a scheduler must enforce under all circumstances the following constraints:

1. *The maximum activation delay when resuming back from idle is limited to $T_R - b_{p,max}$*
2. *As long as a partition p is not idle, the scheduler must provide in each time window of size $\Delta t = T_R$ the maximum budget $b_{p,max}$*

Budgets and replenishment period must be assigned according to (5.2).

At this point we highlight the most important difference between an SPS based mechanisms and TDMA. TDMA always provides service to partition, even if it isn't actually needed as there is no outstanding workload. In case of a SPS based mechanism this is not the case, as only used budget is accounted and replenished later. In general, service provisioning is only needed for partitions which have outstanding workload and are currently not idle. Due to the possibility for a partition to self suspend its execution when idle, a maximum initial delay needs to be considered. In case of TDMA, the delay between an IRQ or task activation and the next time the corresponding partition p achieves service is bounded to $T_{TDMA} - T_p$. This delay occurs when an activation happens right at the end of the corresponding time partition such that it can not be processed anymore. An example of such a behavior has already been shown in Figure 4.2. For a SPS based mechanism, this must be considered separately, as the self suspend introduces an additional partition state compared to TDMA. As long as a partition is not in the idle state, the activation delay is enforced through the TDMA-like service provisioning. When a partition is idle and gets resumed by any kind of activation, it is not necessary to schedule the partition immediately. Instead, only the activation delay needs to be enforced, which means that the partition is delayed by a maximum of $T_R - b_{p,max}$. While this enables a more flexible scheduling, it must be ensured that this can never be violated. Simultaneously to the activation delay the service provisioning must also be ensured. As an example, if a partition p is delayed initially for its maximum of $T_R - b_{p,max}$, it must receive afterwards

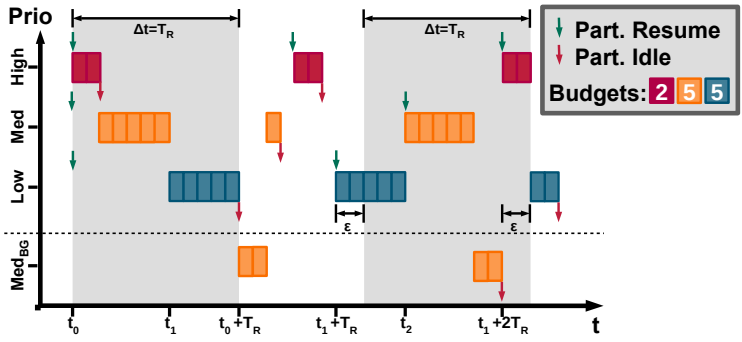


Figure 5.2: Service provisioning according to POSIX

its entire budget of $b_{p,max}$ without preemption if needed. Otherwise, partition p would not receive its maximum budget $b_{p,max}$ within a time window of size T_R.

With Definition 5.1 in mind, a new scheduler can be developed. While this chapter focuses on the general architecture and the RTA, we postpone the actual scheduler according to Definition 5.1 to Chapter 7. But before we introduce the RTA, we will first test if the standard POSIX implementation with a priority based scheduling is able to fulfill Definition 5.1. Let us consider a setup with three partitions with different priorities as shown in Figure 5.2. The priorities for background scheduling are arranged in the same order as the foreground priorities. According to the POSIX definition, the foreground priority of a task is always higher than the background priority. For the sake of simplicity, background scheduling is only used for the medium priority partition (Med/Med_{BG}) in Figure 5.2. The three different partitions have maximum budgets of $(2, 5, 5)$ according to Figure 5.2. A partition is resumed, when one of the tasks inside is activated. The idle state is reached when all tasks inside a partition have served all of their outstanding activations.

In order to check if the POSIX definition of the SPS satisfies Definition 5.1, activation delay and maximized service is checked for the partition with the lowest priority (*Low*). Let us consider the critical instant at t_0 where all partitions are resumed at the same time. Due to the higher priorities, the low priority partition is delayed until t_1. As all other partitions already depleted their budgets, the low priority partition can execute until T_R. Even though the medium priority partition has outstanding workload, it is delayed until T_R where it is scheduled based on its background priority. This simple example shows, that the priority based scheduler as proposed in POSIX is able to enforce the maximum activation delay, even for the critical instant.

The second constraint of Definition 5.2 requires that all partitions receive their

maximum budget within a time window of size $\Delta t = T_R$ as long as they're not idle. We highlighted two corresponding windows in the example from Figure 5.2. The first window from $t_0 \rightarrow T_R$ includes the previously discussed example and satisfies, beside the maximum activation delay, also the required service. The second window from $t_1 + T_R + \epsilon \rightarrow t_1 + 2 \cdot T_R + \epsilon$ shows a different behavior. Although the low priority partition is not idle, it does not receive its maximum budget within the shown time window. In order to show how this is possible let us start at $t_1 + T_R$. Here the low priority partition is resumed and dispatched immediately as all other partitions are idle at $t_1 + T_R$. The low priority partition executes until its budget is depleted at t_2. As the partition still has outstanding workload, it does not switch back to idle. At t_2 also the medium priority partition is resumed, immediately dispatched and starts execution until the budget is depleted. Now the scheduler switches to background scheduling, as both partitions have outstanding workload but no remaining budget. In our example, also the background priority Med_{BG} is higher than Low_{BG} (not shown in Figure 5.2). Therefore, the execution of the medium priority partition is continued until $t_1 + 2 \cdot T_R$. This on its own wouldn't be a problem, as the low priority partition received its maximum budget during $t_1 + T_R \rightarrow t_1 + 2 \cdot T_R$. In order to continue to do so, the low priority partition must be dispatched at $t_1 + 2 \cdot T_R$, where also its depleted budget is replenished. But in Figure 5.2 this is not the case, as the high priority (*High*) is resumed right at $t_1 + 2 \cdot T_R$ and immediately dispatched due to its higher priority. As a result, the received budget for the low priority partition during $t_1 + T_R + \epsilon \rightarrow t_1 + 2 \cdot T_R + \epsilon$ is lower than the corresponding TDMA-bound. A simple measure to avoid such a behavior is, that a non-idle partition is always dispatched immediately when budget is replenished. The second constraint of Definition 5.1 would be violated otherwise. In case of the POSIX conform implementation, this is not possible due to the scheduling with fixed priorities. Again, as this chapter focuses more on the formal aspects, we postpone the actual scheduler implementation to Chapter 7. Nevertheless, for the RTA we assume that the scheduler's behavior is according to Definition 5.1 and implements the previously mentioned measure which provides immediate service after replenishment for certain partitions.

5.1.3 Work conserving scheduling

One of the major requirements from Section 2.3.1 is a work conserving scheduler. We already mentioned that this is not the case for the standard TDMA based scheduling in ARINC653. If we take a look at the previous example from Figure 5.2, we can see that due to the background scheduling the system never delays unnecessarily the execution of a partition. As an example, at $t_0 + T_R$ the medium priority enters background scheduling, since it still has outstand-

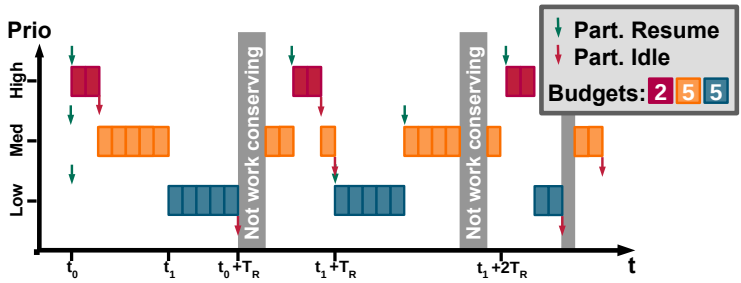


Figure 5.3: Service provisioning according to POSIX without background scheduling

ing workload but its budget was depleted at t_1. This mechanism leads to a work conserving scheduling, since a partition can be picked by the scheduler for dispatching as long as it has outstanding workload, even with an empty budget. For a scheduler without the support for background scheduling this is different as shown in Figure 5.3. While the partitions are resumed at the same time compared to Figure 5.2 the scheduler does not pick them for dispatching if there is no budget left. Therefore, at $t_0 + T_R$ the medium priority partition is not scheduled and the system is idle although there is still outstanding workload. This is again the case later in the example between $t_1 + T_R$ and $t_1 + 2 \cdot T_R$. This shows, that an SPS based system is only work conserving if the scheduler implements also a background scheduling strategy which utilizes unnecessary idle times.

5.2 WCRT Analysis

As already mentioned in Section 4.4 we will provide different definitions for the blocking term $B_{p,i}(\Delta t)$ from (4.3). The different blocking terms are named according to their scheduling techniques.

$B_{p,i}^{TSPP}(\Delta t)$: Partition-level TDMA, Task-level SPP

$B_{p,i}^{SSPP}(\Delta t)$: Partition-level SPS, Task-level SPP

$B_{p,i}^{SBSPP}(\Delta t)$: Partition-level SPS + background scheduling, Task-level SPP

For comparison we also list the TDMA based scheduling with $B_{p,i}^{TSPP}(\Delta t)$, which is given via (4.13) and (4.17). The following section provides the corresponding terms for $B_{p,i}^{SSPP}(\Delta t)$ and $B_{p,i}^{SBSPP}(\Delta t)$. Like in (4.14), the final $B_{p,i}(\Delta t)$ is given either as

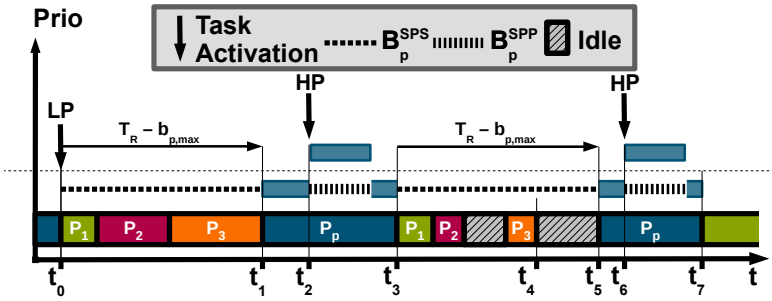


Figure 5.4: SPS based interference without background scheduling

$$B_{p,i}(\Delta t) = B_{IRQ}(\Delta t) + B_{p,i}^{SSPP}(\Delta t) \tag{5.3}$$

or

$$B_{p,i}(\Delta t) = B_{IRQ}(\Delta t) + B_{p,i}^{SBSPP}(\Delta t) \tag{5.4}$$

As described in Section 5.1.2, the POSIX definition is not able to provide a service corresponding to TDMA. In order to achieve the desired service provisioning, the partition-level scheduling does not consider different priorities. In general, all partitions are handled equally and dispatched in a First-In First-out (FIFO) manner to satisfy the targeted behavior form Definition 5.1. This is implemented with a set of different queues. For more implementation specific details we refer at this point to Section 7.1.

First we start with an RTA for the easiest case which is a simple SPS based system without background scheduling. Based on this, the RTA is modified in order to support background scheduling. The formal definition of this RTA allows the integration of different background scheduling techniques (e.g. priority based). Additionally, this section also provides a simplified method, which can be used for a queue based background scheduling.

5.2.1 Without background scheduling

With the previous introduction we will now construct the RTA for an SPS based system with a budget provisioning according to Definition 5.1. Figure 5.4 shows the general interference a task may observe after activation during execution. Like on a system with a partition-level TDMA scheduling, the interference is split into two parts. One based on higher priority activation (HP) inside the same partition and second, the budget provision based on the SPS. Like in (4.13) this leads us to:

$$B_{p,i}^{SSPP}(\Delta t) = B_{p}^{SPS}(\Delta t) + B_{p,i}^{SPP}(\Delta t) \tag{5.5}$$

The example in Figure 5.4 considers a lower priority task (LP) in partition P_p which gets activated at t_0. The SPP based interference is demonstrated with two HP activations of a task in the same partition at t_2 and t_6. Interference based on the SPS is caused by the partitions $P_1, P_2 \& P_3$ and can be seen from $t_0 - t_1$ and $t_3 - t_5$. At this point the drawback of a hard SPS based budget limiting without background scheduling can be seen at t_4. Although outstanding workload does not exist in other partitions at that point in time, P_p is still delayed until t_5 because of a missing budget. As already mentioned before, without background scheduling the SPS mechanism is as little work conserving as the TDMA based scheduling.

The critical instant for the RTA can be constructed with the two following constraints. First, we assume that all higher priority tasks inside the partition are activated at the same time. As a result we can reuse the SPP blocking from (4.16). Second, we consider the activation of the task under analysis right at the point where the entire budget of the corresponding partition has just been used in one block before. This way the activated task achieves the maximum activation delay of $(T_R - b_{P,max})$. Due to the constant service within each $\Delta t = T_R$, the task will always see $(T_R - b_{P,max})$ as interference repetitively, as long as it has not finished execution. As interference, based on the SPS, we get therefore:

$$B_{p}^{SPS}(\Delta t) = (T_R - b_{P,max}) \cdot \left\lceil \frac{\Delta t}{T_R} \right\rceil \tag{5.6}$$

Comparing (5.3) with (4.17) show, that the SPS mechanism introduces the same amount of interference compared to a TDMA based system. Therefore, we observe the same WCRT for a task in both systems, if the SPS configuration is derived from a working TDMA configuration, according to (5.2). Nevertheless, we assume to observe a better average case performance for an SPS based system, if the configuration includes some slack time. The reason for this is the self-suspend mechanism which results in a fragmentation of the budget usage. During runtime this should lead to a lower initial blocking time in the average case.

5.2.2 With background scheduling

Analysing the SPS with background scheduling starts the same way as before. First we divide $B_{p,i}^{SBSPP}(\Delta t)$ into two parts, given as:

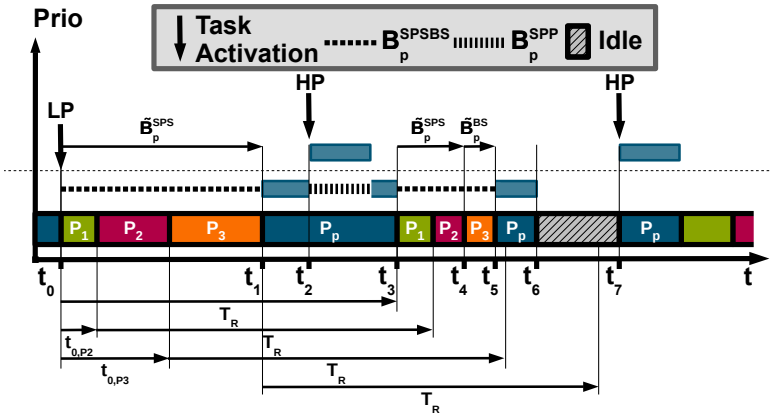


Figure 5.5: SPS based interference with background scheduling

$$B_{p,i}^{SBSPP}(\Delta t) = B_p^{SPSBS}(\Delta t) + B_{p,i}^{SPP}(\Delta t) \tag{5.7}$$

Again, $B_{p,i}^{SPP}(\Delta t)$ is the well-known internal SPP based interference and $B_p^{SPSBS}(\Delta t)$ describes the interference caused by the SPS with background scheduling. In case of a system with background scheduling, the interference must be considered separately for the RTA. On one hand there is the interference caused by the budget replenishment and on the other hand there is the interference during the background scheduling. An example for this is given in Figure 5.5. Like in Figure 5.4, at t_0 a lower priority task inside partition P_p is activated. As critical instant we assume, that P_p just replenished its budget and all other partitions (e.g. P_1, P_2 & P_3) have their entire budget left and outstanding workload. Therefore, all other partitions are scheduled before P_p is dispatched at t_1. Again, like in Figure 5.4 higher priority interference inside of P_p is shown at t_2 and also later at t_7. At t_3 partition P_p is preempted based on an empty budget. Due to $P1$'s budget replenishment (marked with T_R starting at t_0), it immediately starts execution as budget is replenished for a partition with outstanding workload. Same happens for P_2, when its corresponding budget is replenished. At t_4 both, P_1 and P_2 don't have any outstanding workload and stop execution. Both partitions served their workload request only while having a budget available. This type of interference on P_p is marked as $\tilde{B}_p^{SPS}$ in Figure 5.5. It is caused by partitions that serve all their outstanding workload only while having a budget available.

P_3 and P_p still have work to do at t_4 but both partitions do not have a budget as their replenishment lays in the future. At this point the standard SPS would delay both partitions until budget is available, as it was shown in Fig-

ure 5.4. With background scheduling the SPS has the opportunity to dispatch a partition whose budget is currently zero. Within our example in Figure 5.5, both partitions enter background scheduling at t_4/t_5. P_3 is executed first and therefore causes additional interference to P_p, which is not included in the default SPS based interference $\tilde{B}_p^{SPS}$. This interference is marked as $\tilde{B}_p^{BS}$ and depends on the implemented background scheduling strategy. A possible implementation could be a priority or queue based background scheduling. In case of a priority based background scheduling, each partition would have a priority used only during background scheduling. In order to determine the interference it would be sufficient to include all partitions in $\tilde{B}_p^{BS}$ which have outstanding workload, a zero budget left and a higher background priority than the considered partition P_p. For a queue based background scheduling this gets a bit simpler, as $\tilde{B}_p^{BS}$ would then include all other partitions with outstanding workload and a zero budget.

Partition P_p starts executing again at t_5 and finishes the considered activation at t_6. Compared to Figure 5.4 the higher priority activation at t_7 would not cause any interference to the considered lower priority task. Due to the SPS budget enforcement, $\tilde{B}_p^{SPS}$ and $\tilde{B}_p^{BS}$ in combination can never cause more interference than the standard SPS interference without background scheduling (B_p^{SPS} from (5.6)). This leads to:

$$B_p^{SPSBS}(\Delta t) = \min\left\{\tilde{B}_p^{SPS}(\Delta t) + \tilde{B}_p^{BS}(\Delta t),\ B_p^{SPS}(\Delta t))\right\} \tag{5.8}$$

As already mentioned, $\tilde{B}_p^{SPS}$ is the accumulated interference caused by all other partitions ($\Gamma_{HYP} \setminus p$) during their budget consumption. Therefore, it can be represented as a sum over the individual budget consumptions ($\tilde{B}_{p,j}^{SPS}$) from all other partitions, limited by the SPS's replenishment mechanism for individual partition. This gives us:

$$\tilde{B}_p^{SPS}(\Delta t) = \sum_{j \in \{\Gamma_{HYP} \setminus p\}} \tilde{B}_{p,j}^{SPS}(\Delta t) \tag{5.9}$$

Same can be done for $\tilde{B}_p^{BS}$ which represents the accumulated interference during background scheduling from all other relevant partitions. In general $\tilde{B}_p^{BS}$ includes interference caused from other partitions during the background scheduling. As already mentioned this interference highly depends on the scheduling technique used for background scheduling. For a general description we use the term $\mathcal{I}(\Gamma_{HYP} \setminus p)$ to define a set of partitions, that might interfere with partition P_p during background scheduling. As an example, for a priority based background scheduling a background priority is assigned to each partition. $\mathcal{I}(\Gamma_{HYP} \setminus p)$ then contains all partitions with higher or equal priority compared to p. The sum over this set gives us the interference for the background scheduling:

$$\tilde{B}_p^{BS}(\Delta t) = \sum_{j \in \mathcal{I}(\Gamma_{HYP} \setminus p)} \tilde{B}_{p,j}^{BS}(\Delta t) \tag{5.10}$$

Calculating the accumulated interference is straight forward. More challenging instead is the determination of the individual $\tilde{B}_{p,j}^{SPS}(\Delta t)$ and $\tilde{B}_{p,j}^{BS}(\Delta t)$ for each partition. This can be done based on the dispatch order of the interfering partitions, before the considered partition P_p is scheduled first. Within the remainder of this chapter we will label the considered order as $\vec{\alpha}_{p,y}$. Figure 5.5 shows this for the particular order $\vec{\alpha}_{p,y} = (P_1\ P_2\ P_3\ P_p)$. Based on the order, the interfering partitions might start with an offset relative to t_0. In Figure 5.5 this is shown for P_2 and P_3 with $t_{0,2}$ and $t_{0,3}$. Those offsets are important, as the budget replenishment is always relative to the budget consumption. Resulting from this, a partition j with a larger $t_{0,j}$ will be replenished later, which might influence the interference distribution between $\tilde{B}_{p,j}^{SPS}(\Delta t)$ and $\tilde{B}_{p,j}^{BS}(\Delta t)$. In order to determine this interference distribution we need to know the actual requested workload of an interfering partition P_j. For this purpose we use $\beta_j(\Delta t)$, representing the accumulated requested workload of all tasks in partition P_j (given as task-set Γ_j) during a time window Δt.

$$\beta_j(\Delta t) = \sum_{k \in \Gamma_j} \eta_{j,k}^{+}(\Delta t) \cdot \overline{C}_{j,k} \tag{5.11}$$

In order to construct the budget based blocking within a time window Δt of P_j to the considered partition P_p, the minimum of requested workload and granted budget under consideration of an offset $t_{0,j}$ is needed. This leads to

$$\tilde{B}_{p,j}^{SPS}(\Delta t) = \min\left\{\beta_j(\Delta t),\ \max\left\{0, \left\lceil \frac{\Delta t - t_{0,j}}{T_R} \right\rceil \cdot b_{j,max}\right\}\right\} \tag{5.12}$$

The granted budget is a simple step function, where each step is $b_{j,max}$ tall and T_R wide. The initial offset $t_{0,j}$ then shifts this step function to the right and the *max* against 0 eliminates negative budget values.

Each $t_{0,j}$ of a partition j is calculated based on the partitions which lay in front of j in the considered order $\vec{\alpha}_{p,y}$. For the offset calculation we define two look-up functions:

$f_{F,p,\vec{\alpha}_{p,y}}(j)$: The *Forward* translation delivers the position of partition P_j, when analysing a task in partition P_p under the currently considered partition order $\vec{\alpha}_{p,y}$

$f_{B,p,\vec{\alpha}_{p,y}}(n)$: The *Backward* translation delivers the partition at position n, when analysing a task in partition P_p under the currently considered partition order $\vec{\alpha}_{p,y}$

The calculation is then performed based on a fix point iteration as $t_{0,j}$ appears on both sides of the equation. Like in Section 3.2 the calculation is repeated until a fix point is reached when two consecutive calculations lead to the same result. The first partition in the currently considered order $\vec{\alpha}_{p,y}$ will always have an offset equal 0. The offset of the next partition is based on the offset and the requested workload of the previous partition. Again, the requested workload is upper bounded by the maximum budget of the corresponding partition. This way the offsets are given as:

$$t_{0,j} = \begin{cases} 0, & n = 0 \\ t_{0,m} + \max\left\{\min\left\{\beta_m(t_{0,j}),\ b_{m,max}\right\}, \epsilon\right\}, & n > 0 \end{cases} \tag{5.13}$$

The index n is used to identify the position of the considered interfering partition P_j, based on the previously defined look-up function. In order to identify the preceding partition in the considered order $\vec{\alpha}_{p,y}$, we use the index m which is constructed based on the previously calculated position of P_j and the corresponding look-up function.

$$n = f_{F,p,\vec{\alpha}_{p,y}}(j) \qquad m = f_{B,p,\vec{\alpha}_{p,y}}(n-1) \tag{5.14}$$

For the case $n = 1$, the previous offset would be $t_{0,m} = 0$. The fix point iteration would therefore start with $\beta_m(0)$ which returns based on (5.11) always 0. To enforce this the maximum in comparison to an infinitesimal small ϵ is used. This way the fix point iteration for $n = 1$ would not stop immediately.

The workload of a partition P_j, that might interfere during background scheduling, can easily be determined if the partial workload covered by $\tilde{B}^{SPS}_{p,j}$ is known. $\tilde{B}^{BS}_{p,j}(\Delta t)$ is therefore given as difference between requested workload and the already granted budget included in $\tilde{B}^{SPS}_{p,j}$.

$$\tilde{B}^{BS}_{p,j}(\Delta t) = \begin{cases} 0, & \Delta t \leq t_{0,j} \\ \beta_j(\Delta t) - \tilde{B}^{SPS}_{p,j}(\Delta t), & \Delta t > t_{0,j} \end{cases} \tag{5.15}$$

According to (5.12) and (5.15) a partition P_j does not generate any interference for $\Delta t \leq t_{0,j}$. This might be confusing at first glance, but due to the fix point iteration of the busy-window technique and the additional $q \cdot \overline{C}_{p,i}$ in (4.3), the calculation would not stop at $\Delta t = t_{0,j}$.

We already introduced the vector $\vec{\alpha}_{p,y}$ to define the order in which the partitions get scheduled. $\vec{\alpha}_{p,y}$ is also used to perform the forward and backward

look-ups $f_{F,p,\vec{\alpha}_{p,y}}$ and $f_{B,p,\vec{\alpha}_{p,y}}$. Constant $\vec{\alpha}_{p,y}$ is, that the last partition is always P_p, containing the task for the considered busy-window. This is the case, as we always consider the critical instant, where all other partitions get dispatched beforehand. The overall number of partitions in the system is given as the cardinality of the hypervisor task-set $|\Gamma_{HYP}|$. The number of partitions, interfering with P_p, is therefore given as $|\Gamma_{HYP}| - 1$. Resulting from this, the number of possible permutations for the order is given as $(|\Gamma_{HYP}| - 1)!$. Those permutations can be constructed with several algorithms as shown in [56, 105]. Based on the permutations we define for each partition P_p a $Y \times X$ matrix $\underline{A}_P$ with:

$$X = |\Gamma_{HYP}| \qquad Y = (|\Gamma_{HYP}| - 1)! \tag{5.16}$$

Each row of $\underline{A}_P$ would then contain a possible partition order, indexing those rows for $\vec{\alpha}_{p,y}$ is done via y. For the example from Figure 5.5 this would result in:

$$\underline{A}_P = \begin{bmatrix} P_1 & P_2 & P_3 & P_p \\ P_1 & P_3 & P_2 & P_p \\ P_2 & P_1 & P_3 & P_p \\ P_2 & P_3 & P_1 & P_p \\ P_3 & P_1 & P_2 & P_p \\ P_3 & P_2 & P_1 & P_p \end{bmatrix} \qquad \vec{\alpha}_{p,2} = (P_2\ P_1\ P_3\ P_p)$$

A more generic definition for $\vec{\alpha}_{p,y}$ is then given as:

$$\vec{\alpha}_{p,y} = (a_{y1}\ a_{y2}\ \ldots\ a_{y(X-1)}\ P_p) \tag{5.17}$$

Due to the design of the SPS based budget scheduling, the dispatch order of the partitions is in contrast to TDMA not fixed. All possible combinations included in $\underline{A}_P$ must therefore be checked for each task, located in partition P_p. As a result, for each task in the system the busy-window from (4.3) is performed Y-times with (5.4) as blocking term $B_{p,i}$. Each analysis is performed with a different $\vec{\alpha}_{p,y}$, which might result in a different offset $t_{0,j}$ with a different load distribution between $\tilde{B}_p^{SPS}$ and $\tilde{B}_p^{BS}$ for each interfering partition. In the end this might lead to different worst case response times for different offset vectors. The worst case response time is then given as:

$$\overline{R}_{p,i} = \max_{y=1\ldots Y} \left\{ \overline{R}_{p,i,\vec{\alpha}_{p,y}} \right\} \tag{5.18}$$

For systems where dependencies are known (e.g. based on effect chains), impossible partition orders in $\underline{A}_P$ can be removed and don't need to be considered. Finding those dependencies in existing software is an own complex and still rel-

evant topic. Due to the high complexity we did not address this topic directly in our analysis.

5.2.3 Simplified for queue-based background scheduling

The simplest way of background scheduling is an additional FIFO based queue, which contains partitions without budget but outstanding workload. An entry is read from the queue if the system would be idle otherwise. For such a system not only the implementation is straight forward, but also the RTA can be simplified. For the worst case behavior of a queue we assume that all other partitions with outstanding get dispatched before the considered partition during background scheduling. This way $\mathcal{I}(\Gamma_{HYP} \setminus p)$ would include all other partitions from the system. At this point the partition order does not matter anymore. If only a certain set of partitions interfere during background scheduling, different initial offsets influence which partitions can even cause interference. As we do not exclude an interfering partition in $\tilde{B}_p^{BS}(\Delta t)$, the offsets doesn't matter anymore. It is therefore sufficient to take the minimum of the accumulated workload of all other partitions and the maximum SPS interference $B_p^{SPS}(\Delta t)$).

$$B_p^{SPSBS}(\Delta t) = \min\{\sum_{j \in (\Gamma_{HYP} \setminus p)} \beta_j(\Delta t),\ B_p^{SPS}(\Delta t))\} \tag{5.19}$$

(5.19) does not include any separation of replenishment based blocking and background scheduling. For a queue based background scheduling this is not needed anymore as it doesn't matter if another partition interferes during normal or background scheduling. In both cases it is assumed that our considered partition P_p always suffers from interference. The sum over the requested workload from all other partitions indirectly includes the background scheduling. As the overall interference is only bounded by the maximum SPS based interference $B_p^{SPS}(\Delta t)$, a single partition P_j might request more workload than its own budget allows. Including this interference implies that the outstanding workload of partition P_j is then served via background scheduling. Resulting in an execution delay for the considered partition P_p. Without the need of testing each possible permutation for partition orders, (5.19) provides an efficient upper bound regarding the analysis runtime. Also, as a queue provides a conservative worst-case behavior, (5.19) can be used as an upper bound for other background scheduling algorithms like fixed priority.

	TDMA	TDMA+Shaping	SPS	SPS+BS
Isolation	Full	Sufficient	Sufficient	Sufficient
Suspend	No	No	Yes	Yes
RTA	Simple	Simple	Simple	Complex (Simple)
Optimization	None	IRQs	IRQs & Tasks	IRQs & Tasks
Work conserving	No	No	No	Yes
Expected Response Times				
Worst-case	1.0	1.0	1.0	≤ 1.0
Average-case	1.0	≤ 1.0(IRQs) ≥ 1.0(Tasks)	≤ 1.0	≤ 1.0

Table 5.1: Comparison between SPS and TDMA

5.3 Comparison and expectations

Main objective the proposed SPS based budget scheduling technique is to provide a replacement of the TDMA based partition scheduling from ARINC653. Table 5.1 shows a comparison of the different techniques. First the standard TDMA based scheduling as described by ARINC653 without any modifications, which we use as baseline to compare against. It provides perfect temporal isolation due to its entirely static schedule. Therefore, it does not support any mechanism for a partition-level self-suspend. As shown before, the analysis is based on the well-known busy-window approach with a straight forward blocking term. As already stated we use the standard TDMA as baseline, therefore a 1.0 for the expected response times indicates an identical behavior. Values greater or less than 1.0 indicate worse or better response times. In general for the comparison we assume, that for all an identical configuration in terms of time cycle and time partition sizes are used. Same for the derived replenishment period and budged size according to (5.2).

Second there is the TDMA based partition scheduling with additional IRQ shaping. Chapter 4 showed that this technique provides a sufficient temporal isolation, without any support for partition-level self-suspend. The approach optimizes the IRQ processing through the use of available slack inside a time partition. Nevertheless, if there isn't enough slack for all IRQs available, the worst-case is still the same, compared to standard TDMA. As only IRQs benefit from the scheduler modification, those are even more prioritized than tasks. Resulting from this we expect, that the normalized average-case response times are ≤ 1.0 for IRQs but ≥ 1.0 for tasks because of interposed foreign BHs.

Third there is the SPS based budget scheduling without background scheduling. We showed in this chapter, that a sufficient temporal isolation compared to TDMA can be achieved when considering (5.2) for budget sizes and replen-

ishment periods as well as Definition 5.1 for budget provisioning. The additional partition-level self-suspend allows a fragmentation of the budget usage. As we assume that the partition configuration is derived from a TDMA configuration with additional slack, the probability that a partition uses the self-suspend should be maximized. Therefore, the normalized average-case response times should be ≤ 1.0. Reason for this is the probability, that a partition achieves full interference according to (5.6), is minimized. Since the SPS based mechanism does not distinguish between IRQs and tasks, we assume an improvement for both. Nevertheless, probabilities don't change a worst-case contemplation. And in the end, a SPS based mechanism without background scheduling but a hard budget enforcement still provides the same worst-case behavior and is not necessarily work conserving.

In order to change this, we provide the forth scheduling mechanism which is a SPS based budget scheduling with additional background scheduling. Due to the utilization of idle times, it is possible to provide a work conserving partition-level scheduling while preserving sufficient temporal isolation. Nevertheless, this is only the case when the used background scheduling strategy is work conserving. As shown in Section 5.2.2, the RTA is way more complex compared to the other three scheduling mechanisms. Nevertheless, in case of a queue based background scheduling, the RTA can be massively simplified. Like before, the proposed mechanism does not distinguish between IRQs and tasks. We therefore expect an improvement for IRQs and tasks, considering the normalized average-case response times. With the use of background scheduling, we also expect an improvement for the worst-case behavior, as additional time on the resource can be used without violating the sufficient temporal isolation.

All three proposed modifications from Chapter 4 and Chapter 5 fulfil most of the requirements from Section 2.3.1. Only the proof of an efficient implementation is missing at the moment, which will be faced in Section 7.1. This concludes the theoretical discussion of our proposal for the challenge from Section 2.3.1. Before we discuss the actual implementation of the proposed mechanisms, we first address the second challenge from Section 2.3.2 in the following chapter.

"Allons-y!"

- The Doctor

CHAPTER 6

The LET Paradigm as Coordination Instance

The example from Section 2.3.2 shows that the migration from single- to multicore is challenging. Especially if the existing software, written for singlecore execution, should be reused. Even if dependencies within certain effect chains are known, the example from Section 2.3.2 showed that synchronization and coordination is crucial. While the execution order on a singlecore system is primarily defined by static scheduling parameters, this isn't the case anymore when migrating to multicore due to possible parallel execution. In order to explain this even further, Figure 6.1 shows the example from Section 2.3.2 in a slightly modified representation. For the shown example τ_{10} would always execute prior to τ_{20} on a singlecore setup due to the priority assignment based on RMS. When migrating to multicore, the way how communication between different tasks is established gets more and more important. As discussed in Section 2.2.1 the communication between different tasks is usually performed based on shared memory variables. Within Figure 2.7 this was mentioned as data pre- (I) and post-processing (O). Part of this pre- and post-processing is the reading of input and writing of output data. Usually local copies of global variables are used inside a task. This is done during pre- and post-processing, where local copies from the global variables get generated or are written back. Main objective of this mechanism is the preservation of data consistency. This way all runnables inside a task

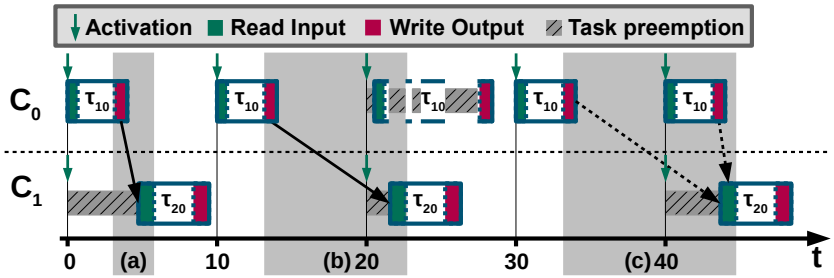


Figure 6.1: Multicore implementation with interference

work on the same input data, even though the global variables are updated in between by a higher priority task or IRQ. Within the context of AUTOSAR this is mentioned as implicit communication.

Figure 6.1 marks the read and write access to the global variables explicitly. Both tasks get activated at the start of period. Compared to τ_{10} which is activated every $10ms$, τ_{20} skips every second period. Again, in case of a singlecore implementation, τ_{10} would always execute prior to τ_{20}, which then reads the most recent input data. If both tasks are distributed to different cores, this implicit order is no longer mandatory. It can be the case if the start of τ_{20}'th execution is delayed after the write access of τ_{10} to the memory has been finished. As shown for the marked region *(a)* in Figure 6.1 this can be the case if higher priority interference delays the execution of τ_{20} on C_1. With such a constellation τ_{20} reads data (marked with a black arrow) which has been written within the same period. In contrast to this, if τ_{20} receives less or τ_{10} more interference, τ_{20} reads input data from the previous period. This is shown during region *(b)*, where the read access of τ_{20} happens prior to the write access of τ_{10}'s instance of the current period. But even if this behavior is not desirable, it can get worse. While in regions *(a)* and *(b)* the read data is consistent, this isn't the case anymore in region *(c)*. Here read accesses of τ_{20} and write accesses of τ_{10} overlap. At this point τ_{20} reads data, which has been written by either the previous or the current instance of τ_{10}. While it is already less than optimal when one input variable was written by the previous and another one by the current instance of τ_{10}, it can be devastating if τ_{20} would read a partly updated variable.

The relevance of data consistency highly depends on the implemented application, but in general it is a desirable goal to always provide consistent data. Also, the susceptibility against a varying data ages depends on the implemented application. Nevertheless, when considering control engineering a constant data age is desirable [90]. The behavior of either *(a)* or *(b)* should therefore be imple-

mented for an application, but not a mixture of both.

A possible mechanism to implement the desired behavior could be based on additional locks. This way a task starts its execution only if the logical preceding task of the effect chain finished its execution. While such a behavior can directly be enforced by the scheduler on a singlecore, a comparable mechanism is not for free on a multicore. Instead, additional communication between multiple cores is needed in order to provide a flexible cross-core locking. While the code overhead for the additional functionality is negligible, the runtime overhead is more important, especially as such a mechanism is either based on inefficient polling or additional IRQs.

Within this chapter we consider a different lock-less method based on the so called Logical Execution Time. The work presented in this chapter is primarily based on the publications [27] and [25] as well as on a contribution to [35, 44]. For the sake of simplicity we describe the integration based on a standard task setup without an additional virtualization. Nevertheless, the described mechanisms should also work for a system with multiple virtualized applications on the same ECU.

6.1 The LET Paradigm

The first reference to the LET paradigm can be found in the origins of the Giotto programming language [61]. General idea of Giotto was to provide an abstraction layer between control applications and the execution environment. It defines time-triggered sensor readings and task activations as well as different execution modes. Access to input or output values is performed through so called *drivers* at specified points in time.

[61] describes the design flow for a Giotto based system as follows. First the control algorithm is designed according to methods already mentioned Section 2.2.1. With the use of code generation, the tested control algorithm is implemented in an application for later integration. Next, a hard- and software independent Giotto description defines the temporal behavior of the designed application. This includes the specification when drivers or applications should be executed. The configuration is passed to the Giotto compiler, which generates a binary file based on the used specification. In order to execute the application according to the generated description, Giotto provides in its execution environment two different components called E-Machine [62] and S-Machine [63]. The E-Machine interprets the generated Giotto binary and coordinates the activation of either drivers or applications. At this point, the E-Machine is responsible for the temporal coordination but not for the actual execution. This is done by the S-Machine which then executes drivers or applications according to the commands

from the E-Machine. In order to map this behavior to our known model, the S-Machine provides the same functionality as an OS, while the E-machine generates task activations. Some more implementation details of the Giotto based design flow can be found in [60].

A by-product of the development of Giotto was the LET paradigm, which abstracts the access to in- and output variables from the actual execution. The idea is motivated by the fact, that the behavior of an application within the real-time domain heavily depends on the actual points in time when in- and outputs are read or written [74]. Therefore, the LET paradigm describes fixed points in time, when in- or outputs are read or written completely independent of the underlying hard- or software. In the context of LET, often two other paradigms are addressed for comparison [73]. The first one is called Zero Execution Time (ZET). The ZET paradigm assumes, that a task (or program) executes in zero time and therefore without any delay between read, execution and write. This model seems to be not realistic at first glance, as a task always needs some time for execution. On the other hand, this model corresponds exactly to the behavior of simulation software like MATLAB/Simulink which we described in Section 2.2.1.

The second paradigm is called Bounded Execution Time (BET) and describes the well-known behavior as covered by classic real-time analysis. A task might need some time after activation to produce a result, but this time is upper bounded to a certain value. This is exactly the behavior described by Section 3.2. Converting the BET to our used system model results in the WCRT. This means, that an upper bound for write accesses at the end of a task exists, but early write accesses based on lower interference than the worst case would not be enforced. The example from Figure 6.1 already showed that such a behavior might cause problems.

As already mentioned the LET paradigm defines explicit points in time for read and write access. It is a mixture of both, ZET and BET. The LET applies fixed points in time for read and write, while allowing an BET behavior for the remaining part of the application. An example for LET is shown in Figure 6.2 with a task τ. The upper half of the Figure 6.2 shows the LET abstraction which

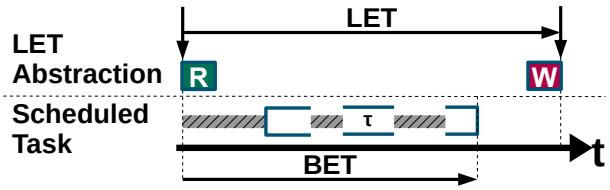


Figure 6.2: Logical execution time of τ

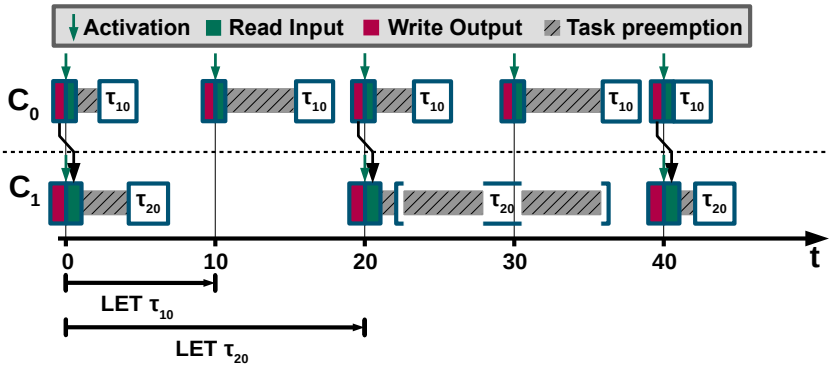


Figure 6.3: Multicore implementation with interference

defines the read access (R) at task activation and the write access (W) after the defined LET. The actual execution of the corresponding OS task shown in the lower half Figure 6.2 depends on the implemented scheduling technique of the OS. According to Figure 6.2, we can directly derive a system requirement for a given abstract LET. The LET must always be greater or equal to the actual BET of the corresponding OS task. Otherwise, it can not be guaranteed that a task finishes its execution in time before the end of the LET. In general, the defined LET should be independent of the actual used hardware and software platform. This means, that a preemption during execution, based on the used scheduling strategy and a set of interferes, doesn't affect the specified read and write times of the LET. Figure 6.3 shows how the introduction of LET may influence the data consistency, when executing on multiple cores. For the sake of simplicity the LET of both tasks is fixed to their periods. Due to the fixed points in time for data input, the mentioned behavior *(b)* from Figure 6.1 is always enforced. Therefore, τ_{20} always reads data which was produced by the instance of τ_{10} from the previous period. Even though this introduces an input delay, it eliminates the input jitter since the input delay is constant.

Based on Giotto, the Timing Definition Language (TDL) was developed [112]. Like Giotto, TDL is also based on a time-triggered behavior and provides an abstraction that maps the LET behavior. Although LET is scheduling agnostic, both approaches are often used in combination with a TDMA based S-Machine. There are several publications available, presenting an integration with different scheduling mechanisms. As an example, the authors of [119] and [27] showed the possibility to use a SPP-based scheduling. The primary difference between both approaches is the granularity on which the LET paradigm is applied. The pro and cons of the different granularities will be discussed later in Section 6.3.

In contrast to the previous integration, the authors in [46] showed, that also EDF scheduling can be applied in the S-Machine. As we consider primarily the automotive domain, an EDF-based scheduling is less relevant than SPP, which is the desired scheduling strategy in OSEK and AUTOSAR OS. In general, it is desirable not to change the current SPP as this would entail major changes in the design process. However, this approach does only allow communication at task boundaries, which leads to long response times on effect chains across several cores. Even in case of a parallel execution this may lead to long end-to-end latencies as shown in [18]. The authors of [48] proposed to keep that schedule and change the role of the LET requirements. Instead of using the WCRT as lower bound for the LET, they change the worst case model of to a typical worst case model allowing shorter LET at the cost of a formally bounded number of LET (i.e. deadline) misses. [119] also justifies this for control loop properties.

Recent work considering LET in the automotive domain primarily target the analysis of effect chains. Primary driver for this idea is the fact that LET removes the input jitter and improves the predicabilty. The industrial motivated WATERS challenge in 2017 [54] compared different effect chain analysis methods for different communication types, including a LET-based shared memory communication. The different contributions [52, 83, 32, 34] showed analytical approaches, for end-to-end timing delays. Some workshop contributions have later been expanded to full conference publications [82, 31]. In general a good overview of currently existing analysis methods is given in [22].

6.2 Lock-less Zero-Time Communication

While the theoretical benefits of LET has been widely discussed, the actual implementation and integration into an existing automotive OS has been largely ignored so far. Only a few publications [98, 31, 46] consider actual implementation aspects. Crucial is the implementation of the timed communication at the LET boundaries, which should be performed in *zero-time* according to the LET paradigm. While providing a robust theoretical framework with atomic data transfers, implementing a zero-time communication is not straight forward. First of all, an execution time equal zero is simply not possible for a data copy operation on a processor. A copy operation always takes some time, even if it is performed via a dedicated DMA controller. Second, the execution for different payload sizes times for read/write accesses vary. As read/write accesses should be executed on a privileged level to ensure data consistency, the entire system is preempted for such accesses. This means, that long execution times for large data chunks to be communicated would represent a not negligible overhead.

In order to provide a zero-time communication, this section describes a simple

mechanism with limited overhead. General idea is to use a double buffering for variables with cross core dependencies. This is based on the fact, that in the automotive domain a classic publisher subscriber paradigm with global variables is used for ECU internal communication, as already mentioned in Section 2.2.1. A variable is therefore only written/updated by one task (publisher), while it might be read by several other tasks (subscribers). The basic functionality of the proposed mechanism operates in the following manner. For each cross core dependent variable a double buffer with twice the memory of the corresponding variable exists. One half of this double buffer is used for read (subscriber) and the other half for write (publisher) accesses. Publisher and subscriber access the double buffer through distinct pointer for write and read access. At the end of each publisher's LET read and write pointer then get swapped. The previous write buffer is now used as read buffer and vice versa.

An example for this mechanism with one publisher and two subscribers is given in Figure 6.4. R and W are the pointers used for read and write access. We use a C-style syntax with an * to indicate the actual access to the memory referenced by the corresponding pointer. Therefore, both rows for R^* and $^*W$ in Figure 6.4 show the actual content of the referenced memory, which changes over time due to write accesses or pointer swaps. In contrast to the behavior defined by the LET paradigm, the actual memory accesses performed by the tasks are still performed bounded to the actual execution of the task. Due different interference, the memory access is not fixed to certain moment in time. Nevertheless, because of the double buffering, including pointer indirection and swap, it occurs to all subscribers as if the publisher acts according to the LET paradigm. The memory access can therefore be performed in the same context of the corresponding task and does not need to be treated separately. In order to achieve this effect, the mechanism needs to enforce *early write* and *delayed read* accesses.

The enforcement of an early write behavior is shown in Figure 6.4. The publisher τ_0 is executed on C_0, the two subscribers τ_1 and τ_2 on C_1 and C_2. For the sake of simplicity we omit the read accesses during τ_0 and write accesses during τ_1/τ_2, as those are not relevant for the shown example. The publishing task τ_0 starts execution and finishes with a write access. From $t_0 \ldots t_1$ τ_0 writes c to the buffer through the dereferenced write pointer $^*W$. As the LET of τ_0 has not yet been finished, the published variable must not be visible to other subscribing tasks. Next, τ_1 starts execution and accesses the referenced address from the read pointer $^*R$ at t_2 and reads b from the read buffer. Although new data has already been written at t_1 it is not visible for a subscriber until t_3, which is compliant to LET paradigm. At t_3 read and write pointer get swapped and therefore the read pointer now references the data buffer that contains c. If subscriber τ_2 now accesses the read buffer at t_4 it reads the new data c which was written at t_1 and published at t_3. The other buffer, holding data b, is now referenced by the

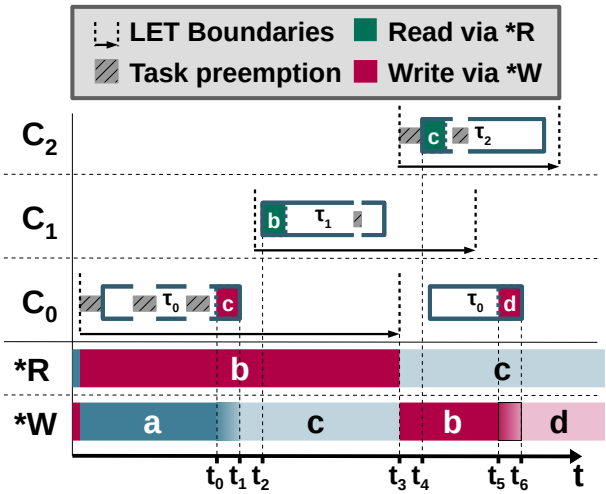


Figure 6.4: Zero-Time Communication early write enforcement

write pointer and gets therefore overwritten at the next write access of τ_0 from t_5 ... t_6. Like in Figure 6.4, the tasks might also get preempted in Figure 6.2. As long as they are able to finish their execution, including the needed time for read and write accesses, everything is fine. The zero-time communication implementation with a double buffering is therefore entirely scheduling agnostic, as required by the LET paradigm.

The first implementation of this mechanism mentioned in [27] only enforced an early write behavior, which can be sufficient for certain system configurations. This means, the system does not care when the actual data is written by a publisher, as the zero-time communication is performed with the pointer swap. Nevertheless, this simple measure is not sufficient if LETs of different tasks overlap. According to the definition of the LET paradigm, the read access is performed directly on task activation. We already mentioned for the previous example in Figure 6.4 that also the read access is performed in the task's context. Due to interference from other tasks or additional IRQ handling, this read access might not be performed directly at task activation, While this isn't a problem in the example from Figure 6.4, this changes if the actual task execution is delayed beyond the LET boundary of the publishing task. Figure 6.5 shows an example for this behavior.

Again the example in Figure 6.5 uses a publishing task (τ_0) on C_0 and two subscriber on C_1 (τ_1) and C_2 (τ_2). At t_0 τ_0 finishes its execution and changes the

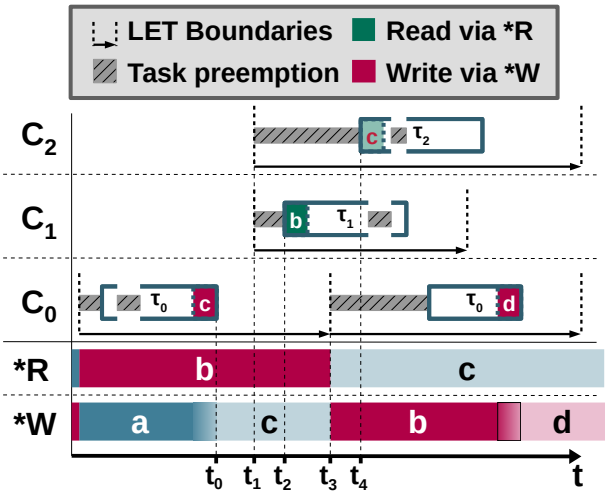


Figure 6.5: Zero-Time Communication delayed read enforcement

data referenced by $^*W$ to c. The LET of both subscribing tasks τ_1 and τ_2 start at t_1. Both tasks are initially delayed. τ_1 performs therefore a read access at t_2 and reads b, which is the correct data value according to the LET paradigm. At t_3 the LET of τ_0 ends and the previously written data is published. When τ_2 now accesses the read buffer at t_4 it reads the value c published at t_3. This is wrong according to the LET paradigm, as the LET of τ_2 has started while the read buffer contained b. In order to fix this, it is sufficient to create a read pointer backup at the start of the corresponding LET (e.g. at t_1). This way τ_2 would access the correct buffer at t_3. In a system where the publishers period is way shorter than the period of a subscriber, also a simple double buffer is not sufficient. Due to the publishers short period, previously written data would simply be overwritten too soon.

Figure 6.6 shows how the previously described behavior can be enforced with a simple ring-buffer implementation storing more than two version of a variable at a time. Let us assume, that τ_1 and τ_2 read data, written by τ_0 and the ring-buffer *rb* is able to store four entries. τ_1 is activated with an offset relative τ_0, but the period (and in this example also the LET) of both is still the same. As a result, τ_1 reads each value, written by τ_0 in the previous period. In contrast to this τ_2 has a three times longer period than τ_0. Therefore, during the LET of τ_2, τ_0 produces three new output values which need to be stored at different locations in order to preserve the desired delayed read behavior. In Figure 6.6, the LET of τ_2 starts at t_1.

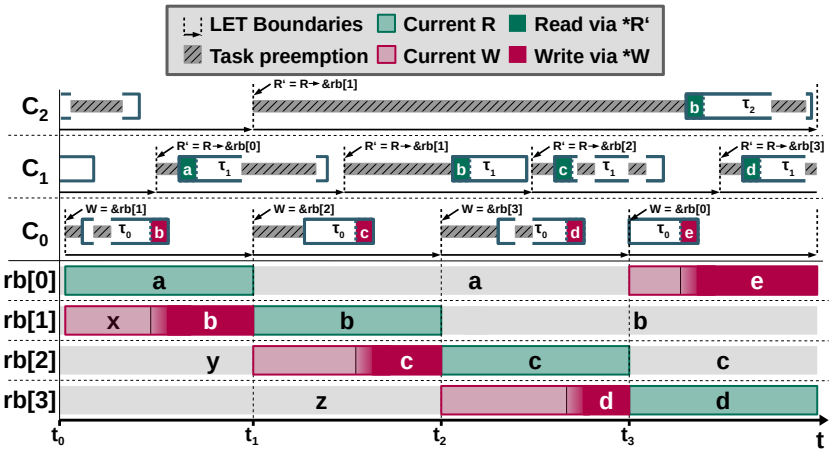


Figure 6.6: Zero-Time Communication based on ring buffer

The relevant input value from τ_0 was therefore produced between t_0 and t_1. In our example τ_0 writes *b* to the ring-buffer entry *rb[1]*, which becomes visible to τ_2 at t_1. At t_1 the LET of τ_2 begins and an internal copy (*R'*) of the current read pointer (*R*) is generated. Within the shown example the execution of τ_2 is delayed due to high priority interference on C_2. Due to the local read pointer copy (*R'*) and the multiple entries in the ring-buffer, τ_2 still reads the correct value (*b*) when it starts execution. Even though τ_0 had already produced new output data at t_2 and t_3, the desired delayed read behavior according to the LET paradigm is ensured this way.

While enforcing a delayed read behavior is comparatively simple, it is heavily application dependent if this is even necessary. For some applications in control engineering it would be preferable to always take the most recent value as input data. Otherwise, a phase shift might be introduced, leading to possible instability of the controlled system [90]. Therefore, although the LET paradigm dictates implicitly when data shall be read, it should be optional for an application if a delayed read behavior is enforced or not. Multiple LETs can also be aligned in way, such that an explicit delayed read enforcement mechanism is not needed. Again this is heavily application dependent. Therefore, the user has the opportunity to decide if an enforcement is used or not in the later explained implementation.

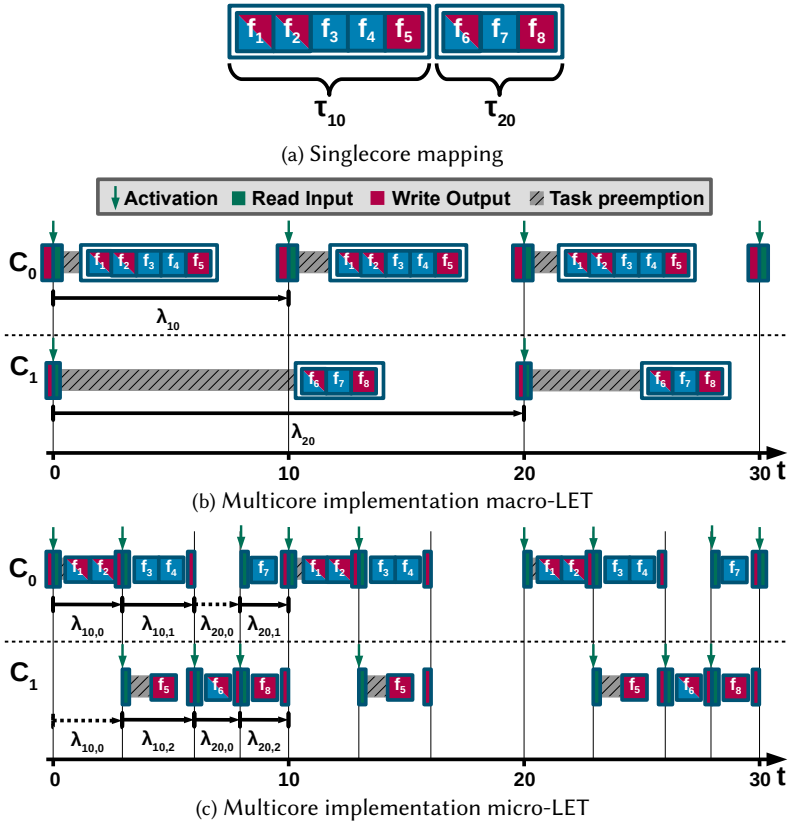
6.3 LET in Automotive Software

The LET paradigm has already been considered by several automotive OEMs and software suppliers [44] to be a useful coordination instance. Nevertheless, there are different views on how LET can be applied to existing software. In this context there are two primary points of contention. First the software granularity on which LET should be applied and second the temporal dimensioning of LET sizes. Resulting of both are two major interpretations of LETs applicability to automotive domain.

The first interpretation directly applies LET to existing software with the aim of achieving maximal flexibility. As a result, the LET paradigm is directly applied on existing tasks without further optimization regarding offsets or parallelizability. With an LET equal to the tasks period the system setup does not change from an OS perspective which is explained in this section. This interpretation is primarily driven by the authors of [119] and will be referred as macro-LET in the remainder of this document. The majority of the related work from [52, 83, 32, 34, 82, 31, 22] is based on this LET interpretation. The second interpretation applies LET on a smaller granularity to an optimized schedule. Result is a use-case optimized schedule, with much smaller LETs. This second interpretation is proposed primarily by the authors of [59] and will be referred as micro-LET in the remainder of this document. Even though, both interpretations are based on the same LET paradigm, major differences, especially regarding the implementation, exist.

One major difference between macro and micro LET, is the way how existing application tasks are mapped to LET tasks and different cores. At this point it is important to point out that we distinguish between actual application tasks τ_i scheduled by an OS and LET tasks λ_i. In our context a LET task is only a temporal frame that implicitly defines the communication (read input, write output). An application task is, as already mentioned in Section 2.2.1, implemented as a set of runnables executed in a static order. Often, the runnables can be grouped into basic blocks, which only share intermediate data dependencies.

Considering the mapping between application and LET tasks, it is straight forward in case of macro-LET. For each application task only one LET task is used. According to [119] an already existing application task from a singlecore implementation should not be executed in parallel on different cores. Instead, entire application tasks might be placed on different cores. In some cases it might make sense to spilt an existing application task to multiple cores, but this is usually to be avoided due to unknown dependencies in existing legacy software. Both, mapping to LET tasks and multicore distribution is therefore preferably performed on a task granularity. In contrast to this the micro-LET approach proposes a parallel execution of independent basic blocks. Therefore, LET tasks are mapped to



(a) Singlecore mapping

(b) Multicore implementation macro-LET

(c) Multicore implementation micro-LET

Figure 6.7: Comparison of LET task mapping for macro- and micro-LET

smaller basic blocks instead of entire tasks. This implicates that one application task might be mapped to multiple LET tasks. [59] describes how such a mapping based on a singlecore implementation can be performed.

In order to explain the differences even further Figure 6.7 provides a simple example. First, Figure 6.7a shows the original tasks used for the singlecore implementation, with two different effect chains. The first chain is $f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 \rightarrow f_6 \rightarrow f_7$ (blue) and the second one is $f_1 \rightarrow f_2 \rightarrow f_5 \rightarrow f_6 \rightarrow f_8$ (red). Therefore, both chains share the runnables f_1, f_2 and f_6. Figure 6.7b shows how this example could be mapped under the consideration of macro-LET. The previously used tasks are distributed to different cores and the LET of each task is set equal to its period. In contrast to this Figure 6.7c shows the mapping under the consideration of micro-LET. Instead of one big LET task, both application

tasks have been divided into three smaller basic blocks according to the desired dataflow based on the effect chains, with a distinct LET task per basic block. As a result, independent basic blocks like (f_3, f_4) and f_5 can be executed in parallel on different cores. Due to the smaller distribution granularity based on the acquired basic blocks, the corresponding LETs are also shorter.

When designing LET based systems, the end of a task's LET is conterminous to a task deadline. In order to check the correctness of a designed system it must be ensured, that each task is able to finish execution during its LET. In case of macro-LET nothing changes compared to the previous system setup, as the defined LETs are equal to the task's periods. Since the previous system setup was RMS based, nothing changed with respect to task deadlines. Therefore, the system can simply be analyzed with the existing RTA methods. Only the task sets on distinct cores differ from the previous setup due to the multicore execution.

In case of micro-LET this is a different, as basic blocks and the corresponding LETs are much shorter now. An additional requirement for the micro-LET approach according to [59] is, that the execution of all LET managed application tasks must fit into the smallest period. For the example in Figure 6.7 this means, that the execution of τ_{10} and τ_{20} must be finished within the period of τ_{10}. Otherwise, the assignment of the shown micro-LET slots would not be possible. Since all LET-based tasks (e.g τ_{10} & τ_{20}) are now synchronized, they don't interfere anymore on the same resource. Even though the micro-LET slots might look like a TDMA based system, it isn't the case. The micro-LET slots do only define a temporal region for execution without any impact on the actual scheduling policy.

We already mentioned that a task's WCRT denotes a lower bound for a possible LET. The only difference in case of micro-LETs is that the WCRT is calculated for a smaller execution time with a reduced set of interferes. Analyzing a micro-LET based system is then straight forward. Regarding the example from Figure 6.7c, this leads to

$$\begin{aligned}
\overline{C}_{\tau_{10,0}} &= \overline{C}_{f_1} + \overline{C}_{f_2} &\rightarrow \quad \overline{R}_{\tau_{10,0}} &\leq \lambda_{10,0} \\
\overline{C}_{\tau_{10,1}} &= \overline{C}_{f_3} + \overline{C}_{f_4} &\rightarrow \quad \overline{R}_{\tau_{10,1}} &\leq \lambda_{10,1} \\
\overline{C}_{\tau_{20,1}} &= \overline{C}_{f_7} &\rightarrow \quad \overline{R}_{\tau_{20,1}} &\leq \lambda_{20,1}
\end{aligned}$$

for C_0 and to

$$\begin{aligned}
\overline{C}_{\tau_{10,2}} &= \overline{C}_{f_5} &\rightarrow \quad \overline{R}_{\tau_{10,2}} &\leq \lambda_{10,2} \\
\overline{C}_{\tau_{20,0}} &= \overline{C}_{f_6} &\rightarrow \quad \overline{R}_{\tau_{20,0}} &\leq \lambda_{20,0} \\
\overline{C}_{\tau_{20,2}} &= \overline{C}_{f_8} &\rightarrow \quad \overline{R}_{\tau_{20,2}} &\leq \lambda_{20,2}
\end{aligned}$$

for C_1. We define for each core C_i two sets of LET tasks. The first set $\Lambda_{i,\epsilon}$ includes

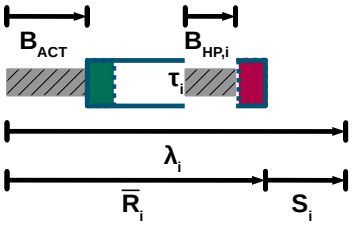


Figure 6.8: Interference during execution inside LET boundaries

all LET tasks used for the execution of basic blocks on core C_i. The second set $\Lambda_{i,\delta}$ includes all LET tasks used to offset basic blocks for synchronous execution on different cores. For the example from Figure 6.7c this results in:

$$\Lambda_{0,\epsilon} = \{\lambda_{10,0}, \lambda_{10,1}, \lambda_{20,1}\} \qquad \Lambda_{0,\delta} = \{\lambda_{20,0}\}$$
$$\Lambda_{1,\epsilon} = \{\lambda_{10,2}, \lambda_{20,0}, \lambda_{20,2}\} \qquad \Lambda_{1,\delta} = \{\lambda_{10,0}\}$$

In order to check schedulability for the constructed schedule the sum over $\Lambda_{i,\epsilon}$ and $\Lambda_{i,\delta}$ must be smaller or equal than the used minimum period on each core

$$\sum_{\lambda \in \Lambda_{i,\epsilon}} \lambda + \sum_{\lambda \in \Lambda_{i,\delta}} \lambda \leq \min_{\tau_j \in \Gamma_{i,LET}} P_j \tag{6.1}$$

where $\Gamma_{i,LET}$ contains all application tasks, executed and decoupled within LET boundaries.

Even though there are obvious differences between macro- and micro-LET, the RTA is still the same. Both approaches can be analyzed with the busy-window from Section 3.2. The only differences are as mentioned above the mapping granularity and the interfering task-set. In general, the execution of a task or basic block follows the example from Figure 6.8. After activation, the execution of a task initially delayed, based on the LET implementation. This activation delay is marked as B_{ACT} in Figure 6.8. Additionally, also blocking based on higher priority interference ($B_{HP,i}$) needs to be accounted. At this point $B_{HP,i}$ can be based on higher priority task or additional IRQs for peripheral hardware. The generic blocking $B_i(\Delta t)$ from Definition 3.15 can then be replaced with:

$$B_i(\Delta t) = B_{ACT} + B_{HP,i}(\Delta t)$$

While $B_{HP,i}$ depends on the actual task-set, B_{ACT} highly depends on the implementation framework. Therefore, the evaluation in Section 8.2.1 shows how an accurate value can be determined for the proposed implementation framework from Section 7.2.

	Macro-LET	Micro-LET
LET task mapping	$1x\tau \rightarrow 1x\lambda$	$1x\tau \rightarrow nx\lambda$
LET sizes	$\lambda = P$	$\lambda \leq P$
Input jitter	0	0
End-to-end latencies	$\gg \ldots \geq$ Singelcore	$\leq$ Singelcore
Scheduling slack	Usually large	Usually small
Susceptibility under increased load	Potentially low	Potentially increased
Flexibility	Flexible	Static (use-case optimized)

Table 6.1: Comparison table of macro- and micro-LET

Table 6.1 shows a comparison of macro- and micro-LET. First of all, the primary differences between both approaches are the task mapping and the LET sizes. While macro-LET proposes a direct mapping between application and LET tasks with LET sizes equal to the task periods, micro-LET proposes a much finer granularity with much shorter LET sizes. As a result of this different LET sizes, such systems behave different in many ways. First, the end-to-end latencies of effect chains. The example from Figure 6.7 shows directly the difference between both regarding end-to-end latencies. In Figure 6.7b τ_{10} and τ_{20} get activated at $t = 0$. Even though τ_{20} on C_1 starts executing after the LET of first the instance of τ_{10} on C_0 finishes at $t = 10$, the freshly created output value will be discarded. Instead, an older value from $t = 0$ is used during the execution of τ_{20}. Compared to a singlecore execution where τ_{20} would start execution right after τ_{10} has finished (due to RMS based scheduling), the end-to-end latencies with macro-LET may increase. This has already been observed in [52]. In contrast to this, the micro-LET approach allows an optimized basic block placement with different offset. This way, the same end-to-end latency compared to a singlecore implementation can be achieved. Additionally, if possible the parallel execution of independent blocks allows to shorten the end-to-end latency even more.

Second and third, the available scheduling slack and resulting from this the susceptibility under increased load. In case of macro-LET this is straight forward, as nothing really changed compared to the singlecore implementation except that less interfering tasks may share the same core. As defined in Definition 3.17 the slack is given as difference between deadline and response time. Since deadline, period and LET are equal in case of macro-LET, the slack stays more or less the same compared to a singlecore implementation. It's even more likely that the amount of slack grows, since interference from higher priority tasks might be removed due to execution on a different core. Resulting from the potentially increased slack, a macro-LET based system is more robust and tolerates more additional load from unpredictable sources (like peripheral IRQs). The maximum

amount of additional allowed load/interference can be calculated on the normalized slack during execution within LET boundaries. The minimum available normalized slack $\hat{S}_{min}$ over all available LET tasks defines the maximum allowed interference.

$$\hat{S}_{min} = \min_{i \in \Gamma} \frac{\overline{R}_i}{\lambda_i} \tag{6.2}$$

Without any information about the additional interference (e.g. if it is aligned with the scheduling) this must apply to any time window of size $\Delta t = \lambda_i$. This limitation for additional load is also valid in case of micro-LET. Due to the shorter basic blocks and LET sizes the absolute amount of slack decreases. Even normalized to the LET sizes, this amount of slack in a micro-LET based system is usually smaller compared to macro-LET. Primary reason for this is the LET task mapping which optimizes the end-to-end latencies. As already mentioned, this implicates that all LET tasks must be able to finish within the smallest period of the application tasks. In general this leads to a smaller normalized slack and therefore to a system which is much more prone to deadline (or LET) misses when dealing with additional loads from unpredictable sources. From the available slack and the susceptibility under increased load directly the system flexibility can be derived. One of the primary constraints during the design of the original LET paradigm [74] was to create hardware and OS agnostic application software which easily can be migrated to a completely different execution platform. Therefore, in the original LET proposal changing the execution environment must have no effect on the actual LET sizes. While the integration of LET in the automotive domain does not directly aim for maximal independence between application software and execution platform, it makes sense to be aware about the limitations of macro- or micro-LET regarding the system flexibility. Based on the previous observations, macro-LET usually provides a much more flexible system due to additional slack. Therefore, it is more likely that in case of a macro-LET based system it is much more straight forward to change/add software parts or modify/change the execution environment, without any modifications on the LET sizes. In contrast to this, a micro-LET based system is much more static due to the optimized schedule resulting in less slack and flexibility.

Even though, macro- and micro-LET show larger differences, both approaches can be realized with the proposed zero-time communication with either double or ring buffering. Important is in both cases that the implementation provides a low and constant runtime overhead. The memory architecture of the used μC/SoC gets also important, when considering the placement of double or ring buffers. Those aspects depend highly on the actual implementation and will therefore be faced in the following chapter.

"So, no more running. I aim to misbehave."

- Malcom Reynolds

CHAPTER 7

Implementation Challenges

Implementation challenges of mechanisms like the proposed SPS based budget scheduling or the LET synchronization are often neglected in the academic world. Reason for this is the very time-consuming implementation based on complex hardware and comprehensive software frameworks. Since implementation heavy papers are usually hard to publish, the additional effort of a "real-world" implementation is often avoided. While this is absolutely understandable for a PhD student with limited time, it still leads to a gap between academic research and actual problems in the industry.

When designing schedulers or other OS mechanisms, there are several limited resources on embedded systems which need to be considered. As an example, the discussed mechanisms from the previous chapters need both an accurate time base. An operating system itself usually provides a periodic system tick. The granularity of this system tick is often in the range of $1ms$. Since the system tick is based on a dedicated hardware timer, an IRQ is generated for each tick. Therefore, much smaller granularities than $1ms$ are often avoided due to high IRQ loads. For the discussed mechanisms from previous chapters, a granularity of $1ms$ might not be sufficient. As a result, additional timer hardware is needed.

We already mentioned in Section 2.1 that PWM is widely used in embedded systems for different purposes. As result, the generation of such signals is done

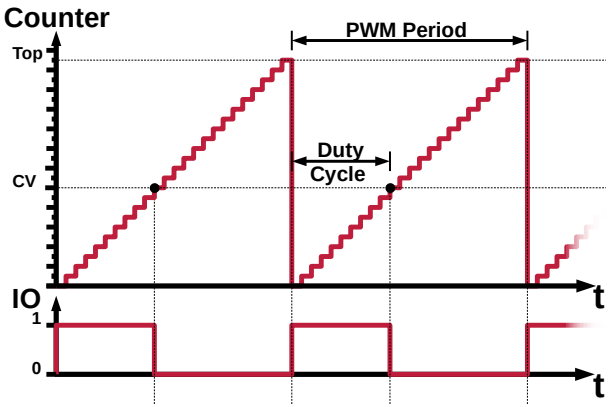


Figure 7.1: Generation of a PWM signal

by distinct peripheral in nearly every modern μC/SoC. The way how the signal generation is achieved is more or less identical in all cases and is shown in Figure 7.1. A free running timer (often also mentioned as counter) counts upwards to a certain *Top* value, wraps around to zero and continues operation. In some cases the top value can be specified by an additional register, but often the top value is given by the maximum bit size of the timer's hardware implementation. The time it takes to count from zero to the top value is mentioned as *PWM Period* which can be varied by the timers frequency and/or the top value itself. In order to generate a PWM signal an additional register is used which contains a compare variable *CV*. The actual state of the output *IO* is then coupled to a compare operation, e.g. as long as the current timer value is smaller than *CV* the output is set to 1, otherwise it is set 0. Changing *CV* does than directly influence the PWM *Duty Cycle*.

Beside the PWM generation, such timers are usually multi functional. The output signal generation can be usually disabled and the timer can be configured to generate an IRQ when the timer value is equal to *CV*. This way a free running timer can be used to generate highly accurate timed IRQs. Additionally, they can be used for fine granular time measurement. The maximum time that can lay between two consecutive generated IRQs or can be measured between two events, is in both cases limited to the timers period (*PWM Period*). Modern μCs/SoCs like the AURIX or R-Car H3 provide multiple timers with such a functionality, often with more than one compare register. Nevertheless, it is desirable to use such hardware timers as efficiently as possible in order to still provide further timer modules to the application.

Listing 7.1: Task state check with if

```
1  int check_taskstate(TCB_T *pTcb)
2  {
3    if(pTcb->State == READY)
4      return 0;
5    if(pTcb->State == RUNNING)
6      return 1;
7    if(pTcb->State == SUSPENDED)
8      return 2;
9    if(pTcb->State == WAITING)
10     return 3;
11
12   return -1;
13 }
```

Listing 7.2: Task state check with switch

```
1  int check_taskstate(TCB_T *pTcb)
2  {
3    int ret=0;
4    switch(pTcb->State)
5    {
6      case READY:     ret = 0;  break;
7      case RUNNING:   ret = 1;  break;
8      case SUSPENDED: ret = 2;  break;
9      case WAITING:   ret = 3;  break;
10     default:        ret = -1; break;
11   }
12   return ret;
13 }
```

Beside the limited peripheral resources, also some basic rules should be considered when developing mechanisms targeted for the implementation in small RTOSs. In general a small overhead regarding code size, data and runtime overhead is desirable. While a small runtime overhead for an OS is advantageous, a constant runtime overhead is much often more important. A constant runtime overhead is easier to integrate in a RTA and leads overall to a much more predictable system behavior. In case of OSs, task state dependent decision taking is a major functionality. A simple example for such a decision is shown in Listing 7.1. With several *if*-statements the state of a task is checked based on its TCB and the function returns with a corresponding value. If the task is in an invalid state, an error (−1) is returned. The functionality of such a function is straight forward and the code is easy to understand. Nevertheless, in case of a RTOS an implementation like in Listing 7.1 should be avoided since the runtime depends on the task state. Instead of an implementation based on multiple *if*-statements, a *switch* based solution like in Listing 7.2 is desirable. A *switch*-statement is usually compiled to a lookup based jump table, which results in a constant runtime for each state decision. Even though the simplicity of the used example, such state based decisions are widely used, also in case of entire state machines. Awareness of the discussed runtime variation between both implementation methods is therefore a key feature when implementing such RTOS mechanisms.

In some cases a constant runtime overhead in an RTOS mechanisms is hard to achieve. This may be based on the mechanisms itself, but often a constant runtime overhead is exchanged against flexibility. As an example, during the evaluation we will use two different RTOSs with SPP based scheduling called μC/OS-II [68] and ERIKA OS [51]. While the entire design of μC/OS-II is optimized for this kind of scheduling, ERIKA OS can be configured for several other scheduling strategies (e.g. EDF). This difference directly influences the way how the SPP scheduling is implemented in both RTOSs. In case of μC/OS-II a highly optimized mechanism is used, which implements the SPP scheduling with a constant runtime overhead. In contrast to this ERIKA OS uses a flexible system based on ready queues. A ready queue contains all tasks which are ready for execution and the dispatcher simply reads and executes the task referenced by

the first queue entry. The only thing, which differs between several scheduling strategies is the function which inserts new tasks to the queue. In case of a priority based scheduling the first entry of this queue always contains the task with the highest priority. While the overhead when reading a task from the queue is constant, adding a new task to the queue produces a variable overhead due to sorting inside the queue.

Therefore, ERIKA OS exchanges predictability against flexibility, when comparing it against μC/OS-II. The argumentation how this can be tolerated in a RTOS is, that the maximum runtime overhead has an upper bound. This is the case for the described ready queues from ERIKA OS since the number of tasks inside the system does not change over runtime (and is limited by the OSEK and/or AUTOSAR OS definition). Regarding the constant runtime overhead this means, if a constant runtime overhead can not be ensured during runtime by implementation, it should be upper bounded during compiletime through a parameter. With this general ideas in mind, we will now discuss the implementation of the previously described scheduling mechanismsm from Chapter 4, Chapter 5 and Chapter 6. The work presented in this chapter is therefore based on [26, 24] and [27, 25, 7].

7.1 SPS based Budget Scheduling

Figure 7.2 shows the proposed SPS based architecture, which was already mentioned in Section 5.1.1. It consists of two major parts, first the SPS for interference enforcement and second the scheduler for service provisioning. The interaction between both is realized with a CB API, which is explained further in Section 7.1.2. The SPS has four incoming signals, which primary define the runtime behavior. Those signals are, as already discussed in Section 5.1.1, *Empty*, *Refill*, *Idle* and *Resume*. *Empty* and *Refill* are IRQs based on a timer module, which is under the control of the SPS. The interrupts indicate either an empty budget or a budget replenishment/refill. Therefore, the proper configuration of the timer module is the main component to enforce interference. Due to the importance of the timer module, we explain this component further more in Section 7.1.1.

When a partition served all its outstanding activations, an *Idle* signal can be sent to the SPS. The way how we realized this signal is based on an additional system call. This system call is issued by a partition when it enters its idle loop or idle task. Depending on the actual hardware, system calls may be executed differently. Nevertheless, a system call is always handled within a specific context of the CPU. If this is realized with a distinct context only for system calls or within a shared context (e.g. shared with hardware IRQs) depends on the actual used CPU architecture. For the sake of security it is important that the system

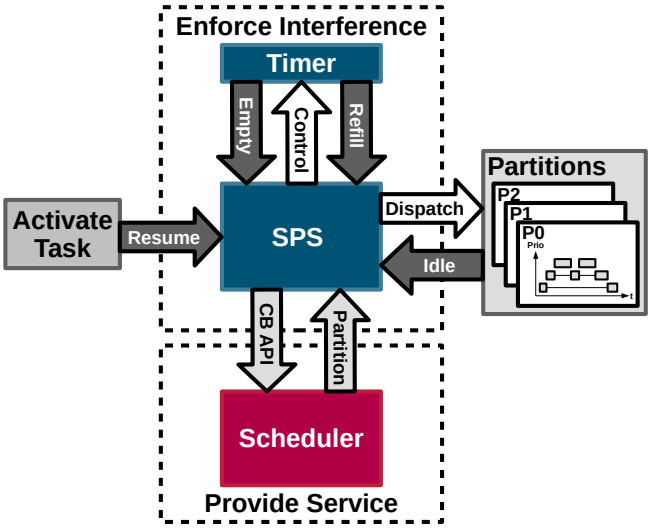


Figure 7.2: Overall architecture for SPS based scheduling

call is always handled in a different context than the actual execution of the partition. This is also the case for the *Resume* signal, which can be issued from an IRQ (e.g. time tick) or a system call (e.g. inter-partition communication).

When handling each of this four signals, the SPS forwards the signal to the scheduler. The decision which partition should be dispatched next, is therefore up to the scheduler. Beside the CB API, Section 7.1.2 will also show how a simple TDMA scheduling can be realized with the proposed architecture from Figure 7.2. Based on this simple example, Section 7.1.3 then explains the actual budget based scheduling.

7.1.1 Timer usage

We already explained how a common PWM timer can be used for a flexible and accurate IRQ generation. Each time the timer reaches the compare value, stored in the dedicated register, an IRQ is generated. Figure 7.2 already showed, that the generation of the budget *Empty* and *Refill* signals is based on a timer module.

The generation of the budget empty signal is quite obvious. Each time a partition is picked for execution, an *irq* must be generated when the budget depletes. Since we use a free running timer (32-bit), which overflows to zero at its defined boundary the compare value for the register can be calculated as:

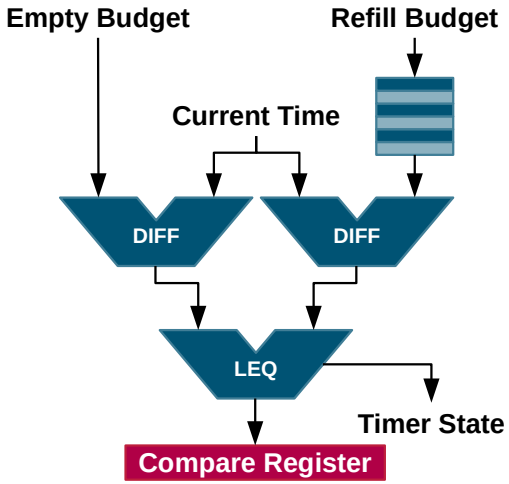


Figure 7.3: Timer compare register configuration

$$CompareRegister = CurrentTime + BudgetLeft$$

Since all values are treated as 32-bit unsigned integers, an overflow is handled automatically. The generation of the budget refill signal is also simple. Each time a partition stops execution, the used budget is prepared for replenishment. As an example, when a partition executes from $t_0 \ldots t_1$, the compare register is set to

$$CompareRegister = t_0 + T_R$$

in order to generate an IRQ at the correct point in time.

A straight forward way to generate both signals would be to use two timers with two compare registers. Since the synchronization of two timers is sometimes challenging, a solution with one single timebase is often preferable. Figure 7.3 shows how both signals can be generated with only one timer and a single compare register. As inputs the absolute time of the current budget empty event and the head of a budget refill queue are taken. While the absolute time of the next budget empty event depends on the currently executed partition, the queue holds all times for future budget refill signals from partitions which have been executed in the past. The queue is therefore ordered according to the absolute times, with the closest refill event at the head and the latest event at the tail. Each time a timer IRQ is issued or the partition context is switched, the following lookup is performed. First, the difference between the current time and

both, next budget empty and next budget refill event, is calculated. Second, the smaller value is loaded to the compare register and an internal timer state is set. The timer state indicates, if the occurred IRQ should send an *Empty* signal, *Refill* signal or both to the SPS core module. Dependent on the sent signal the corresponding signal is then handled by the SPS, which is explained in more detail later in Section 7.1.2.

We mentioned before, that the refill queue holds all times for future budget replenishments and is sorted accordingly. From the idea, this is comparable with the previously discussed ready-queue and an EDF based scheduling. Reading an entry from the queue is therefore performed with a constant overhead, but the insertion into the queue might be challenging. For our system this is not the case as a new entry can always be appended at the end of the queue without violating the chronological sorting. In order to explain this, let us assume the following example with three partitions. The first partition executes from $t_0 \dots t_1$, the second from $t_1 \dots t_2$ and the third from $t_2 \dots t_3$ with $t_0 > t_1 > t_2 > t_3$. According to the SPS the used budgets are replenished at:

$t_0 + T_R$: $t_1 - t_0$ time units for the first partition

$t_1 + T_R$: $t_2 - t_1$ time units for the second partition

$t_2 + T_R$: $t_3 - t_2$ time units for the third partition

Because of the same replenishment period T_R for all partitions in the system, the order of replenishments is always identical to the previously seen execution order. This makes our approach much easier to implement compared to the standard SPS based scheduling, where multiple replenishment periods are possible. Reason for this is, that in case of different replenishment periods it is not valid anymore to always append entries to the end of the refill queue. Instead, the entire queue might be processed in order to find the corresponding place for the ne entry. Alternatively, a distinct timer for each replenishment period could be used to overcome this issue. In reality this implementation based limitation of the original SPS was one reason, why the SPS has seen so little attention in the past. For our proposed system with only one replenishment period, the implementation isn't a problem anymore.

7.1.2 SPS CBAPI

The CB API is the connection between scheduler and SPS. It consists out of a set of primary and secondary callback functions. Each callback function returns a partition which should be scheduled next (except the $CB\_Init()$ callback). The primary callbacks are directly coupled to the ingoing signals *Refill*, *Empty*, *Resume* and *Idle* as already shown in Figure 7.2.

Primary callback functions

CB_Empty(p): Partition p depleted its budget

CB_Refill(p): Budget for partition p was replenished

CB_Idle(p): Partition p finished all outstanding activations and is now idle

CB_Resume(p): Partition p was reactivated because due to an activation of a partition-internal task

Secondary callback functions

CB_Init(): Initialization routine at system startup

CB_NewP(p): Partition p has been added to the system

CB_Reset(p): Scheduler reset during execution of partition p

Beside the four primary, there are also three secondary callback functions which are primarily used during system start-up or reconfiguration. The *CB_Init* function is called during system initialization and allows the scheduler to initialize its internal data. Since the hypervisor implementation, used for evaluation, supported the subsequent loading of partitions, the *CB_NewP* functions is used to inform the scheduler about the newly available partition. In order to reset the internal scheduler states during execution, we also added the *CB_Reset* function. While the secondary callback functions are primarily used for start-up and reconfiguration, the primary callback functions actually define the schedulers budget provisioning. For a better understanding, we will now describe the general conditions under which the primary callback functions are called. The code is presented in Listing 7.3, 7.4, 7.5 and 7.6.

We start with Listing 7.3 which shows the handler for the budget *Empty* signal. Since the handler is called by the timers ISR, the current timer value is passed to the handler. Also, the context of the preempted task is saved on a distinct stack. Depending on the underlying hardware, this is automatically done on IRQ occurrence or must be done by hand in the low-level ISR. First of all a critical section is entered, such that the handler is executed non-preemptive. Next, the used budget of the preempted partition is pushed to the timers refill queue. The two global variables *curID* and *StartTime* hold an identifier and the latest dispatch time of the previously executed partition. The difference between *TimerVal* and *StartTime* therefore indicates the consumed budged of the previously executed partition *curID*. Next the schedulers *CB_Empty* function is called, with *curID* as call parameter. The partition, returned by the callback, is stored in *parID*. If a callback function returns −1, the SPS uses this time to execute hypervisor internal housekeeping functions. If *parID* differs from *curID*, the context of

Listing 7.3: Budget empty handler

```
1  void Budget_Empty(unsigned int TimerVal)
2  {
3    EnterCritical();
4
5    PushUsedBudget(curID,StartTime,TimerVal);
6
7    int parID = CB_Empty(curID);
8
9    if(parID != curID)
10   {
11     SuspendPart(curID);
12     Schedule(parID);
13   }
14
15   ExitCritical();
16 }
```

Listing 7.4: Budget refill handler

```
1  void Budget_Refill(unsigned int TimerVal)
2  {
3    EnterCritical();
4    REFILL_T *pEntry = GetRefillEntry();
5    P_Tbl[pEntry->ID].Bud += pEntry->Bud;
6    int parID = CB_Refill(pEntry->ID);
7    if(parID != curID)
8    {
9      PushUsedBudget(curID,StartTime,TimerVal);
10     SuspendPart(curID);
11     Schedule(parID);
12   }
13   else if(curID == pEntry->ID)
14     ReloadBudgetToTimer(curID);
15   ExitCritical();
16 }
```

the previously preempted partition *curID* is removed from the stack and stored (*SuspendPart*). The context of the new partition *parID* is restored and placed on the stack (*Schedule*). The function call *Schedule(parID)* also updates the global variables *curID* and *StartTime*. At the end of the critical section the timer ISR returns and restores the context of partition placed on the saved stack.

Next we have a look at the handler for the budget *Refill* signal which, like the previously described *Empty* handler, is executed in the context of the timer ISR and receives the current timer value as call parameter. The handler is shown in Listing 7.4 and again the entire handler is executed within a critical section. Since the timer module does not hold any information about size of the replenished budget or the corresponding partition identifier, we store this information separately in a queue. As a result, the needed information is obtained first and a reference is stored in *pEntry*. The reference holds the replenished budget *pEntry->Bud* for partition *pEntry->ID*. Next, *pEntry->Bud* is added to the available budget of the corresponding partition inside the partition table *P_Tbl*. After this is done, the CB_Refill function is called and the context, saved on the stack, is changed if the returned partition (*parID*) differs from the actual preempted partition (*curID*). If the context isn't changed and the budget replenishment is for preempted partition (*curID == pEntry->ID*), the newly available budget is loaded to timer in order to postpone the budget empty signal accordingly. Again, at the end of the critical section, the handler returns to the timer ISR and restores the saved partition context.

The handler for the *Resume* signal is shown in Listing 7.5. Like the previously described timer based signals, the resume signal can also be issued by an IRQ. As an example, a partition can be activated by a peripheral driver in order to process received data. Also, a partition can be resumed based on the system tick. In both cases, the signal handler is executed in the context of an ISR. Additionally, it is also possible to resume a partition remotely from another partition, which is usually achieved through a service call or software IRQ. Like an IRQ, a system call is processed in an own context with a saved context of the preempted partition

Listing 7.5: Partition resume handler

```
1  void Resume(int actID)
2  {
3    EnterCritical();
4    int parID = CB_Resume(actID);
5    if(parID != curID)
6    {
7      unsigned int TimerVal = GetTimer();
8      PushUsedBudget(curID,StartTime,TimerVal);
9      SuspendPart(curID);
10     Schedule(parID);
11   }
12   ExitCritical();
13 }
```

Listing 7.6: Partition idle handler

```
1  void Idle(void)
2  {
3    int parID = CB_Idle(curID);
4    if(parID != curID)
5    {
6      EnterCritical();
7      unsigned int TimerVal = GetTimer();
8      PushUsedBudget(curID,StartTime,TimerVal);
9      SuspendPart(curID);
10     Schedule(parID);
11     ExitCritical();
12   }
13 }
```

on a distinct stack. Changing the context, in order to schedule a new partition, is therefore identical and does not depend on whether the resume handler is processed during an IRQ or service call. Executing the resume handler is straight forward and again done within a critical section. The identifier of the resumed partition is passed to the signal handler as a call parameter (*actID*) and forwarded to the *CB_Resume* function. Like before, if the returned partition differs from the preempted partition, the context on the stack is changed. Since the signal handler is not called within the context of the timer module, the actual timer value (*TimerVal*) must be acquired through a separate function call. Afterwards the handler function is left and the context of the saved partition is restored.

The partition *Idle* signal handler is shown in Listing 7.6 and is executed within the context of a service call. After entering the signal handler, directly the *CB_Idle* function is called with the current partition as call parameter. If the returned partition differs from the preempted partition, the saved context is switched. Compared to the other three signal handlers, only a critical section is entered if a partition switch should be initiated. Reason for this is the fact, that the corresponding service call is often directly issued from the idle task of the guest partition, without any additional monitoring. Due to the non preemptive execution, each critical section might delay the processing of incoming IRQs. In order to prevent unnecessary critical sections, those are entered conditionally in case of the *Idle* signal handler.

Listing 7.7 shows a simple example implementing a TDMA scheduling, where the TDMA slot sizes are equal to the maximum budgets and the TDMA cycle is equal to the replenishment period. The general functionality is constructed based on two arrays. *NextPLookup* is used to identify the next partition to be scheduled with a simple lookup and therefore implements the static order within the TDMA schedule. *PState* holds the current budget state of a partition. In case of an empty budget the entry is set to -1, otherwise it is equal to the corresponding array index (e.g. *PState[p] == p*). Since TDMA does not support any kind of self suspend mechanism, the *CB_Resume* and *CB_Idle* callbacks simply return the current partition (*p == curID* in case of *CB_Idle*), to keep the current partition context. Via *CB_NewP* the state of partition is initially set for each partition

Listing 7.7: Example implementation of a TDMA scheduler based on the proposed CB API

```
1  static const int NextPLookup[] = {1,2,3,0};
2  static int PState[] = {-1,-1,-1,-1};
3
4  /* Primary callback functions */
5  int  CB_Empty (int p) { PState[p] = -1; return PState[NextPLookup[p]];    }
6  int  CB_Refill(int p) { PState[p] = p;  return (curID == -1) ? p : curID; }
7  int  CB_Resume(int p) { return curID;                                     }
8  int  CB_Idle  (int p) { return p;                                         }
9  /* Secondary callback functions */
10 void CB_Init  (void ) { return;                                          }
11 int  CB_NewP  (int p) { PState[p] = p;  return -1;                       }
12 void CB_Reset (int p) { return 0;                                        }
```

and will only be modified by *CB_Empty* and *CB_Refill*. Since we assume that *CB_NewP* is only called on startup, the function always returns −1 to keep the system within the hypervisors context during system initialization. After all partitions have been added, the scheduling is started with call of *CB_Reset* which refers to the first partition (0) within the TDMA schedule. When the budget of the first partition depletes, the *CB_Empty* function is called which sets the partitions state to −1 and returns the state of the next partition based on *NextPLookup* and *PState*. This is repeated until the budget of the last partition within the TDMA schedule depletes its budget. Due to the dependency between TDMA slots and TDMA cycle, the next timer IRQ will generate both, a *Refill* and an *Empty* signal. Within our SPS implementation we define, that the *Refill* is always handled prior to the *Empty* signal. Therefore, we first handle the *Refill* signal and call *CB_Refill* where the state of the first partition (0) is set. When the *CB_Empty* is called afterwards, the previously replenished partition can be dispatched. This consecutive execution of *CB_Refill* and *CB_Empty* is then continued for each timer IRQ, as long as nothing is changed on the partition setup. If a partition is removed from the setup during runtime, the corresponding entry in *PState* is set to −1 which indicates, that the slot can be used by the hypervisor for house keeping. Whenever this happens, the next timer IRQ will only cause a call to *CB_Refill* since the hypervisor itself does not have a budget. This is handled by the conditional return value of *CB_Refill*. If the current timeslot was used by the hypervisor (*curID == -1*) *CB_Refill* returns the replenished partition since there won't be a subsequent call of *CB_Empty* during the same timer IRQ.

7.1.3 Scheduler implementation

Due to the previous sections we gained the necessary knowledge in order to understand the structure of SPS based budget scheduling. Understanding both, timer module and CB API, provides a solid foundation for the following explanation. We start with the explanation of the scheduler without any background scheduling and provide the needed modifications for background scheduling afterwards. Main target of the following implementation is to satisfy Definition 5.1.

Figure 7.4 shows the partition-state graph of the budget scheduler without background scheduling. Such a state machine exists within the system for each scheduled partition. Only one partition at a time can be in the *Run* state indicating that the partition is scheduled on the corresponding core at the moment. In general, the scheduler needs to enforce two different things as discussed in Section 5.1.2. First, when a partition p is activated and leaves the *Idle* state, it must be ensured that the partition won't be delayed longer than $T_R - b_{p,max}$ until it is scheduled. The second thing is, a partition must always see the same service as for a TDMA scheduler as long as it isn't within the *Idle* state.

In general, the system is constructed around two queues. One, holding partitions with outstanding workload and another one for partitions ready to be scheduled (e.g. their current budget is greater zero). Q_{Resume} stores partitions, which just have been activated and Q_{Run} stores partitions which have been preempted during execution. The order inside the queues is FIFO based and both queues are drained based on the *PopQ* command, which is shown in Figure 7.4 and later in Table 7.1. As only one partition at a time can be scheduled by the SPS, the queues are prioritized and Q_{Resume} is only drained if Q_{Run} does not hold any partitions. If both queues are empty and the *PopQ* command is executed, the entire system is idle (*SPS.idle* in Table 7.1). At this point, a partition with no budget could be executed in the background, which will be discussed later. A partition can only be stored in either Q_{Resume} or Q_{Run}, therefore the number of stored partitions in Q_{Resume} and Q_{Run} can't be greater than overall number of partitions Ω at any point in time. In general Table 7.1 shows the scheduling decisions based on callbacks and internal states. *PopQ* indicates, that the next partition is taken from the queue subsystem (e.g. either from Q_{Run} or Q_{Resume}). p indicates, that the partition, which is assigned to the callback, will be scheduled next. *curID* indicates, that the currently scheduled partition wouldn't be preempted independent of the passed callback parameter p. A general design idea of the budget scheduler is to preempt a running partition only if it is necessary to ensure the sufficient temporal isolation.

When a partition is reactivated and leaves the *Idle* state, it is checked if the partitions budget is greater zero. If this is the case the partition is stored at the end of Q_{Resume}, otherwise this step is delayed with the $Wait$ state until the budget of the reactivated partition is refilled. If we assume that partition p was activated at t_0, the interference p might see up to $t_0 + T_R$ is upper bounded by the SPS to $T_R - b_{p,max}$. The partitions that might cause this interference can either be stored before p in one of the queues (Q_{Resume} or Q_{Run}) or is currently picked for execution and inside the *Run* state. Alternatively, a partition in the *Empty* state might cause interference due to a budget replenishment which forces the scheduler to initiate a context switch. Independent of the other partitions state, the interference is bounded by the SPS to $T_R - b_{p,max}$, even if we assume that all

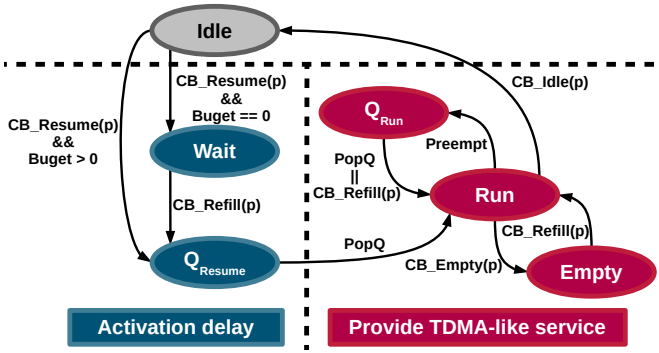


Figure 7.4: Scheduler state machine w.o. background scheduling

other partitions get dispatched prior to the considered partition p. On the other hand, if we consider that p just depleted its entire budget $b_{p,max}$ at $t_0 - \epsilon$ and also entered the idle state, the earliest point in time, when this budget is refilled is:

$$T_{Refill} = t_0 + (T_R - \epsilon - b_{p,max}) \tag{7.1}$$

With $\epsilon \rightarrow 0$ we get the worst case refill time $t_0 + (T_R - b_{p,max})$ for p between t_0 and $t_0 + T_R$, which would be seen by p if it was reactivated right after depleting its entire budget. Even for this case p would be able to receive its entire budget of $b_{p,max}$ before $t_0 + T_R$. The activation delay of the budget scheduler before a partition enters the *Run* state the first time after leaving the *Idle* state, is therefore bounded to $(T_R - b_{p,max})$, which satisfies Definition 5.1. The behavior of the $CB\_Resume(p)$ callback directly follows the idea to only preempt executing partitions if needed. Only if the SPS would be otherwise idle, the callback loads the next partition via *PopQ*. A side effect of this behavior is that a resumed partition inside Q_{Resume} can directly be executed, if no other partition is stored prior to it inside the queue.

After a partition entered the *Run* state it must be ensured that the received service corresponds to a TDMA scheduled system as long as the partition does not enter the *Idle* state. This means in each time window of size $\Delta t = T_{TDMA} = T_R$, p must receive $b_{p,max}$ service. In order to achieve this, a simple method can be used. As long as a partition is not idle, it must be scheduled immediately when budget is refilled. Otherwise, it can not be guaranteed that the service received within each time-window of size $\Delta t = T_{TDMA} = T_R$ is according to Definition 5.1. This is shown in Table 7.1 as first condition of the $CB\_Refill(p)$ callback, where p is scheduled if it is currently in the Q_{Run} or *Empty* state. Those

	SPS w.o. BG		SPS w. BG	
	Condition	**Next Partition**	**Condition**	**Next Partition**
Empty(p)		PopQ	$Q_{Run}.len == 0$ && $Q_{Resume}.len == 0$	PopQEmpty
			!($Q_{Run}.len == 0$ && $Q_{Resume}.len == 0$)	PopQ
Refill(p)	$p.in(Q_{Run})\|\|p.in(Empty)$	p	$p.in(Q_{Run})\|\|p.in(Q_{Empty})$ || $p.in(RunBS)$	p
	!($p.in(Q_{Run})\|\|p.in(Empty)$) && !$SPS.idle$	curID	!($p.in(Q_{Run})\|\|p.in(Q_{Resume})$ || $p.in(RunBS)$)&&!$SPS.idle$	curID
	!($p.in(Q_{Run})\|\|p.in(Empty)$) && $SPS.idle$	PopQ	!($p.in(Q_{Run})\|\|p.in(Q_{Resume})$ || $p.in(RunBS)$)&&$SPS.idle$	PopQ
Idle(p)		PopQ	$Q_{Run}.len == 0$ && $Q_{Empty}.len == 0$	PopQEmpty
			!($Q_{Run}.len == 0$ && $Q_{Empty}.len == 0$)	PopQ
Resume(p)	$SPS.idle$	PopQ	$SPS.idle$	PopQ
	!$SPS.idle$	curID	!$SPS.idle$	curID

$p.in(X)$: True, if p is in state X
$Q.len$: Number of entries in queue Q
$SPS.idle$: SPS is idle, no partition scheduled at the moment
!, &&, ||: C-style boolean logic

Table 7.1: Callback based scheduling decisions for SPS without and with background scheduling

states show the two different reasons, why a partition stops execution while still having work to do. If the partition is in none of those states and the SPS is not idle, the current partition is not preempted. If the SPS is idle and the replenished partition in none of the discussed states, the next partition is loaded from queue system. A partition enters the *Empty* state during execution, when the budget is depleted and the $CB\_Empty(p)$ callback is issued. The partition starts execution again when new budget is available, based on the $CB\_Refill(p)$ callback. On the other hand, a partition enters the Q_{Run} state during execution, when it is preempted by another partition. This might happen if a $CB\_Refill(p)$ callback was executed for another partition, which was either in the *Empty* or the Q_{Run} state. In order to leave the Q_{Run} state, either the queue must be drained via *PopQ* (possible conditions in Table 7.1) or as mentioned before with $CB\_Refill(p)$. If a partition leaves the Q_{Run} state based on $CB\_Refill(p)$, the corresponding entry is deleted inside the queue. Since a partition is only stored in one queue at most, a reference to this queue entry can be saved in a TCB extension. As the queues are implemented as double linked list, removing an entry is performed with a

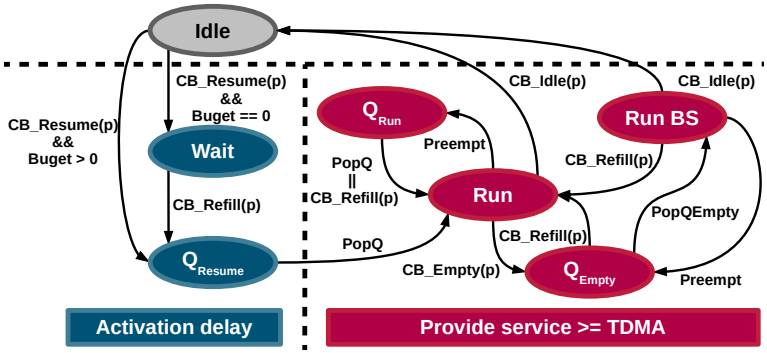


Figure 7.5: Scheduler state machine w. background scheduling

constant runtime overhead. Again, since the interference from other partitions is bounded by the SPS the provided service is according to TDMA, as long as a non-idle partition is scheduled whenever new budget is available. This implies, that a partition p will always get its assigned budget T_p as long as it has work to do, which satisfies Definition 5.1.

When a partition leaves its *Run* state based on the $CB\_Idle(p)$ callback, the next partition is loaded from the queue system via *PopQ*. Compared to the scheduler used in PikeOS [70], service is provided at this point to the next waiting partition in the queue system. PikeOS tries something similar, but instead of assigning processor time to waiting partitions, the time is assigned to the privileged hypervisor partition, which is not necessarily work conserving. Also, if a time partition setup does not include any slack, it is possible that a partition would never enter the *Idle* state and the budget scheduler would work like a TDMA scheduler. Because of this, it is desirable to use an optimization method like the proposed one from Section 4.5 in order to construct valid TDMA-like configurations for the SPS with maximized slack. This way the usage of the $CB\_Idle(p)$ callback can be maximized, which should lead to shorter response times for tasks and IRQs compared to a TDMA scheduler.

The modified partition-state graph for a system with background scheduling is shown in Figure 7.5. Compared to Figure 7.4 a third queue (Q_{Empty}) and an additional state (*RunBS*) hast been added. Each time a partition depletes its budget and does still have outstanding workload, it is inserted into Q_{Empty}. At this point we use Q_{Empty} as a flexible interface for different background scheduling techniques, like the already explained ready queue in ERIKA OS. Since the order inside Q_{Empty} depends on the implemented background scheduling, only the insertion inside $CB\_Empty(p)$ needs to be modified for a different background

scheduling. In our implementation we simply insert the partition at the end of the queue, which implements a simple queue based FIFO scheduling in the background. As described for ERIKA OS, in case of other background scheduling techniques based on e.g. priorities or deadlines a corresponding search must be executed on the entire queue. Q_{Empty} is drained based on the *PopQEmpty* command as shown in Figure 7.5 and Table 7.1. *PopQEmpty* is called each time, the system would be idle otherwise and moves the partition from Q_{Empty} to the *RunBS* state. The partition stays in this mode, as long as it does not self-suspends itself (*Idle(p)*), is preempted by a partition with budget (*Preempt*) or receives a budget replenishment (*Refill(p)*). When budget is replenished for a partition in Q_{Empty}, the queue entry is deleted and the partition is switched back to the usual *Run* state. Like stated before, a partition can only be inside one of the queues at a time, this is also the case with the additional Q_{Empty}.

With look at Table 7.1 it is obvious, that the decision taking gets more complex when implementing background scheduling. During a $CB\_Empty(p)$ callback it needs to be checked if Q_{Run} and Q_{Resume} are empty. If both queues are empty, a partition from Q_{Empty} can be dispatched, therefore entering background scheduling. Otherwise, another partition, stored either Q_{Run} or Q_{Resume}, is dispatched. The exact same conditions are checked for the $CB\_Idle(p)$ callback. In case of $CB\_Refill(p)$, the decision needs to include the *RunBS* state, such that p would be scheduled in regular mode if it was scheduled in the background before. The behavior of the $CB\_Resume(p)$ callback is still the same, compared to system without background scheduling.

In order to keep Table 7.1 understandable, we did not include each implementation specific detail for dispatch decisions. As an example, the implementation must consider the current timer state and the distance to the next timer IRQ. Otherwise, a new partition might be dispatched while a *Refill(p)* callback is issued, overwriting the previously taken decision. As a result, the implementation is more challenging as it might look at first glance, when only considering Figure 7.4, Figure 7.5 and Table 7.1. Nevertheless, such a system can be implemented with a reasonable overhead.

7.2 LET implementation with Zero-Time Communication

While the previous discussed implementation of the SPS based budget scheduling was only minor influenced by actual software standards of the automotive domain, this differs for the implementation of the LET implementation. Major reason for this fact is, that the development of the implementation proposed in this section was influenced by a collaboration with Daimler RD/EIS. As a result of this collaboration the IDA LET Machine (ILM) was released as open source

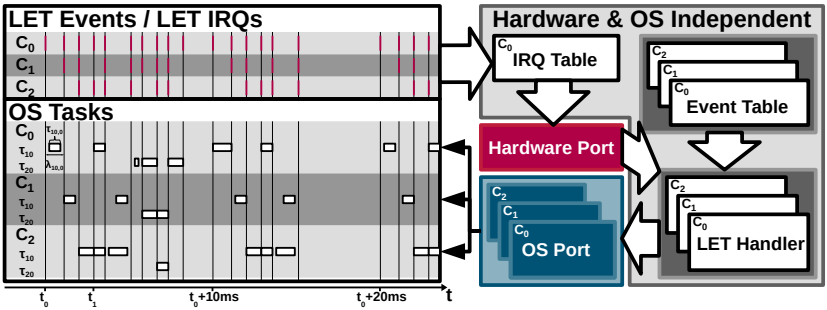


Figure 7.6: Overview of the ILM implementation

software [7] under a GPLv3 with a linking extension which allows the combination with none GPL software. The general structure and the design of the ILM are now discussed in this subsection.

We already introduced the AUTOSAR software architecture in Section 2.2. Since the LET paradigm needs to be tightly coupled to OS and underlying hardware, it should be integrated as a part of the AUTOSAR BSW which also includes the OS itself. Figure 7.6 shows an overview of our implementation proposal, executing a micro-LET use-case. Each LET related action is coupled to a so called *LET Event*. Those actions can be the start of an LET task, a pointer swap or also the backup of a read pointer. As an example, at t_0 in Figure 7.6 the LET task $\lambda_{10,0}$ is started and the corresponding $\tau_{10,0}$ will be executed within $\lambda_{10,0}$. In order to generate the LET events on different cores, we use an *LET IRQ* which is generated on one core by a single timer and then redirected to other cores. In our example, the IRQs are generated on C_0 and redirected to C_1 and C_2 if necessary. If an LET IRQ should also result in an LET event on C_0, it is directly processed afterwards. Therefore, the IRQ pattern generated on C_0 contains events from all other cores and is repeated periodically. The period of the IRQ pattern is defined by the hyperperiod of all LET tasks. As Figure 7.6 contains tasks with a $10ms$ and $20ms$ period, the hyperperiod is equal $20ms$ in this example. From pattern and period we derive an *IRQ Table*. Each entry consists of an offset relative to the hyperperiod and a bitfield which marks all relevant cores for event redirection. The first entries of a table implementing the example from Figure 7.6 may look like Listing 7.8. The table is used to configure a *Hardware Port* which generates the needed IRQ pattern in a most efficient way. This can be done with a timer and/or capture compare unit. Inside the hardware dependent IRQ handler the *LET Handler* on each relevant core is activated, which usually can be done with a software IRQ. While the hardware generated IRQ is only handled on one core

Listing 7.8: Hardware IRQ table for the example from Figure 7.6

```
const ILM_HW_LET_TABLE_T C0_HW_Master_Table[] =                  /* HW table for IRQ generation */
{
  { 0,  (ILM_COREMASK_C0)                                     }, /* LET Event is processed on C0*/
  { 6,  (ILM_COREMASK_C0 | ILM_COREMASK_C1)                   }, /* ... -> C0 & C1               */
  { 8,  (ILM_COREMASK_C1 | ILM_COREMASK_C2)                   }, /* ... -> C1 & C2               */
  { 16, (ILM_COREMASK_C0 | ILM_COREMASK_C2)                   }, /* ... -> C0 & C2               */
  /*...*/
};
```

Listing 7.9: C_0 event table for the example from Figure 7.6

```
const ILM_LET_TABLE_T C0_LET_TABLE[] =                            /* LET event table for C0    */
{
  {0,  C0_T10_0, (ILM_EVENT_TACT), PAYLOAD(0, IN_C0_T10_0_L, 0) }, /* Activate C0_T10_0         */
  {6,  C0_T10_0, (ILM_EVENT_PSWAP),PAYLOAD(0, 0, OUT_C0_T10_0_CS)},/* Pointer swap for C0_T10_0 */
  {16, C0_T10_1, (ILM_EVENT_TACT), PAYLOAD(0, IN_C0_T10_1_L, 0) }, /* Activate C0_T10_1         */
  /*...*/
};
```

(e.g. C_0 in the example from Figure 7.6), the LET handler can be executed on all cores in parallel. Each handler accesses an own LET *Event Table*, which contains again a relative offset, an action to be performed and an additional payload field. An example for such an event table is shown in Listing 7.9. Each entry consists of the relative offset within the hyperperiod and a task identifier. Additional information like the corresponding double buffer of the executed task or the list of input pointers which should be saved on task activation is stored in a generic payload field. In contrast to the IRQ table, the event table might contain multiple entries with the same relative offset. This is the case when two or more actions must be performed "at the same time". An example for this behavior is a simple back-2-back execution of two LET tasks (e.g. on C_2 at t_1), where first the pointer of the previous LET task is swapped and second the following LET task is activated. The activation of an LET task is performed through the *OS Port* abstraction, which maps the LET task activation to an OS task activation. At this point the LET implementation hands over the actual OS task dispatching to OS scheduler. This way the LET implementation activates the OS task via *OS Port* but does not dictate the actual task execution or the scheduling scheme. This way also additional higher priority interference resulting from other non-LET tasks or IRQs can be tolerated and integrated as long the implicit LET deadlines can be met.

In order to integrate Figure 7.6 into an automotive software architecture, a matching layer for each part of the LET implementation should be determined in the AUTOSAR software architecture. This is straight forward for hardware and OS dependent software parts, as those must be located inside the AUTOSAR BSW. The *Hardware Port* should be integrated as driver and the *OS Port* as a system services. For the *Hardware and OS Independent* part several implementation options exist. It can either be integrated as system service or also as an application task.

Listing 7.10: Hardware port API

```
1  /* Master core  IRQ handler */
2  uint32 ILM_HW_Isr_Master(void);
3  /* Slave core IRQ handler */
4  void   ILM_HW_Isr_Slave (uint32 CoreId);
5
6  /* Master core periphery configuration */
7  void   ILM_HW_Init      (ILM_HW_LET_TABLE_T * pTable, uint32 TableSize,
8                           uint32 Freq, uint32 Hyperperiod);
9  void   ILM_HW_StartTimer(void);
```

7.2.1 Hardware Port

The hardware ports purpose of the proposed ILM architecture is primarily the generation of the IRQ pattern and the low-level handling of the corresponding IRQs. In order to achieve this, we use a small API shown in Listing 7.10. We already mentioned that the IRQ pattern is only generated on one core (master), which redirects the IRQs if needed to the other cores (slaves). The interface differs therefore slightly for master and slave cores. For both types an ISR is needed to either handle the IRQ, generated by the underlying hardware timer (*ILM_HW_Isr_Master*) or the redirected software IRQ (*ILM_HW_Isr_Slave*). Even though handling IRQs strongly depends on the underlying hardware, the configuration of the IRQ controller inside the μC as well as managing the different IRQ vectors is under the control of the used OS. This is also assumed in case of the ILM. The OS port must therefore provide the corresponding core identifier when calling the slave core ISR. On the master core only, the ISR return value must be interpreted, in order to execute the ILM event handling afterwards if needed.

Additionally to the IRQ handling, the hardware port must initially configure the underlying peripheral hardware, used for IRQ pattern generation. Therefore, *ILM_HW_Init* is called at startup and initializes the local reference to the used hardware IRQ table (e.g. *C0_HW_Master_Table* in Listing 7.8) and its actual length. Based on the used hyperperiod and minimum frequency (based on the offset granularity) the underlying hardware timer is configured. When the system is configured on all involved cores, the timer subsystem can be started via *ILM_HW_StartTimer*.

We already explained the general timer functionality inside a μC used for PWM signal generation. Like in Section 7.1, the functionality of such a free running timer can be exploited to generate the required IRQ pattern of an LET schedule. Instead of only one compare during the PWM period, we use several compare values (cv) and a PWM period equal to the hyperperiod of the LET schedule. Figure 7.7 shows an example for this behavior, where the generated LET and executing OS tasks are shown under the graph. Instead of a pin toggle during register match, *ILM_HW_Isr_Master* is called in order to reprogram the compare register to the next relative LETs offset (marked as cv_x in Figure 7.7) inside the hyperperiod and forward the IRQ to the slave cores (if required). The

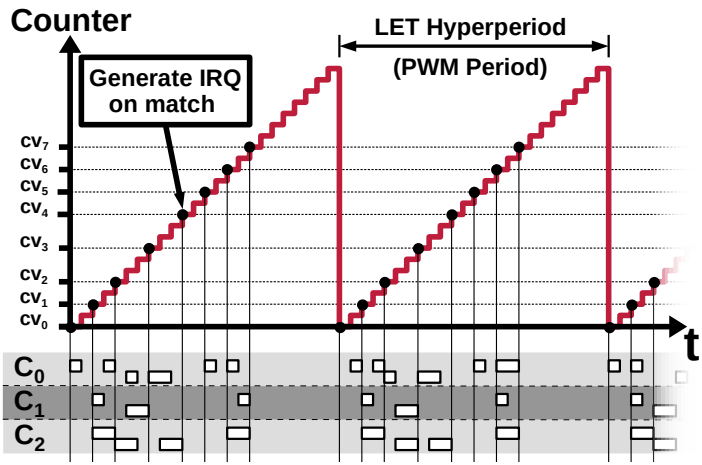


Figure 7.7: Timer configuration

compare values (shown as $cv_0 \ldots cv_7$ in Figure 7.7) are directly derived from the IRQ table (e.g. *C0_HW_Master_Table* in Listing 7.8) with

$$cv_x = LIRQ_x \cdot Scale \tag{7.2}$$

where $LIRQ_x$ is the corresponding entry in the IRQ table. *Scale* is based on the actual frequency of the underlying hardware timer and used to calculate appropriate compare values. In order to provide a sufficient granularity *Scale* must always be ≥ 1.0.

As already mentioned, the compare mechanism is available in most modern μC architectures. This is also the case for the prevalent Infineon AURIX μC family. Section 2.1.1 described that most of the AURIX μCs support the generation of PWM signals in multiple different ways. Either based on the CCU6 or on the GTM. The hardware port of the ILM uses the CCU6 module, due to the simplicity of the implementation. Using the GTM for the same task instead is also possible, but more challenging due to its complexity. It might also lead to resource conflicts since the GTM is the favored hardware module for timing critical applications (e.g. engine control).

7.2.2 OS Port

As shown in Figure 7.6 one major task of the OS port is to map the start of an LET to an application tasks activation within the OS. Exactly this functionality is achieved with *ILM_OS_ActivateTask* as part of the OS ports API shown in

Listing 7.11: OS port API

```
/* Task activation and state check */
void   ILM_OS_ActivateTask   (ILM_TASKS_T TaskId);
void   ILM_OS_SetEvent       (ILM_TASKS_T TaskId, void *pEvent);
uint32 ILM_OS_CheckTaskState (ILM_TASKS_T TaskId);

/* Synchronous task context reentry*/
void   ILM_OS_Set_BarrierMask(ILM_COREMASK_T CoreMask); /* Only on master core */
void   ILM_OS_SyncCores      (void);                    /* On all cores        */
```

Listing 7.12: Master core IRQ handler based on ERIKA OS

```
ISR2(ILM_OS_Isr)
{
#if ILM_USE_HOOK_START_IRQ > 0
  ILM_HOOK_START_IRQ();
#endif
  if(ILM_HW_Isr_Master() == 1)
  {
    ILM_Handler();
    ILM_OS_SyncCores();
  }
#if ILM_USE_HOOK_STOP_IRQ > 0
  ILM_HOOK_STOP_IRQ();
#endif
}
```

Listing 7.13: Slave core IRQ handler based on ERIKA OS

```
ISR2(ILM_OS_Isr)
{
#if ILM_USE_HOOK_START_IRQ > 0
  ILM_HOOK_START_IRQ();
#endif
  ILM_HW_Isr_Slave(EE_CURRENTCPU);

  ILM_Handler();
  ILM_OS_SyncCores();

#if ILM_USE_HOOK_STOP_IRQ > 0
  ILM_HOOK_STOP_IRQ();
#endif
}
```

Listing 7.11. A second function called *ILM_OS_SetEvent* is used to resume the execution of a task waiting for an event. This might be the case, if the controlled task is equivalent to the previously discussed OSEK ECC tasks from Section 2.2.2 and Figure 2.9. Third, the function *ILM_OS_CheckTaskState* is used to obtain the current state of an executing OS task. If a task is running or ready to be executed, the function returns 1, otherwise 0. In general this function is used to check if an OS task finishes its execution before the end of the corresponding LET. If the implicit deadline is missed, a user defined hook function can be called.

We already mentioned, that the configuration of IRQ sources is usually under control of the operating system. As an example, Listing 7.12 and Listing 7.13 show, how this is performed in an OSEK compliant OS like ERIKA OS. The *ISR2* macro defines the function name and registers the ISR to the OS during compilation. Additionally, several user-defined hook functions can be used to implement additional tracing or other stuff if needed. According to the behavior described in Section 7.2.1, the event handling is executed on the master core conditionally. In order to enable a synchronized reentry into the task context, a synchronization barrier is used after the execution of each event handler (*ILM_Handler*). How such a synchronization barrier is implemented depends on the used OS as well as the hardware capabilities. Checking the synchronization barrier is therefore part of the used OS and must be implemented via *ILM_OS_SyncCores* as a part of the OS port. The configuration of this barrier must be according to the cores, which execute LET events for the corresponding LET IRQ. This step is performed inside *ILM_HW_Isr_Master*, since the bitfield used for IRQ redirection also represents the synchronization barrier configuration for the corresponding LET IRQ. The needed functionality for configuration must be implemented via

Listing 7.14: Direct mapping to BCC

```
1  void Task_i(void)
2  {
3    Call_t_i();
4    TerminateTask();
5  }
```

Listing 7.15: Grouped mapping to ECC

```
1  void Task_i(void)
2  {
3    Call_t_i_0();
4    WaitEvent(Event_t_i);
5    ClearEvent(Event_t_i);
6    Call_t_i_1();
7    WaitEvent(Event_t_i);
8    ClearEvent(Event_t_i);
9    /*...*/
10   WaitEvent(Event_t_i);
11   ClearEvent(Event_t_i);
12   Call_t_i_n();
13   TerminateTask();
14 }
```

Listing 7.16: Grouped mapping to BCC

```
1  void (*LUT_t_i[]) (void) = {
2    Call_t_i_0,
3    Call_t_i_1,
4    /*...*/
5    Call_t_i_n
6  };
7  uint16 CNT_t_i = 0;
8  uint16 N_t_i = sizeof(LUT_t_i)/sizeof(void *);
9  void Task_i(void)
10 {
11   LUT_t_i[CNT_t_i]();
12   CNT_t_i = (CNT_t_i + 1) % N_t_i;
13   TerminateTask();
14 }
```

ILM_OS_Set_BarrierMask. In general, a synchronized return to the tasks' context is needed to ensure, that all tasks start their execution after the pointer swaps have been performed. Otherwise, it would be possible that a task on one core starts execution before the pointers of a remote task, previously executed on a different core, has been swapped. As a result, a wrong input value would be read.

Even though the API defines the connection between ILM and OS, it is still an open question how to perform the mapping between LET and OS tasks. The described BCC and ECC task models from Section 2.2.2 are based on the OSEK OS definition and have later been adapted for AUTOSAR OS. Regardless the fact, that both standards are directly tailored to the automotive domain, the definition of ECC tasks also map to many other RTOSs like μC/OS-II. Since this dissertation targets primarily automotive systems, the following task mapping solutions are based on the capabilities of the BCC and ECC task models. The first possible integration is a direct mapping from LET tasks to distinct BCC tasks. This means that for each LET task λ_i an own OS task τ_i is used. This works for macro-LET tasks as well as the basic block based micro-LET scheduling. Each time the LET handling starts a LET task λ_i, the OS port calls *ILM_OS_ActivateTask* in order to tell the OS to schedule τ_i as soon as possible. Listing 7.14 shows an example for such direct mapping. The function call *Call_t_i* represents the execution of τ_i. In case of a micro-LET this would correspond to Call_t_i_n and $\tau_{i,n}$ for each basic block.

While a direct mapping is the only possible mapping for a macro-LET, this differs in case of a micro-LET based system. As already mentioned in Section 6.3, micro-LET based synchronization is usually applied to basic function blocks which have been scheduled in container tasks before. Based on the previous system design, the resulting LET tasks often have common periods but different offsets. Because of this, it is possible to map a set of micro-LET tasks which share the

same period to one single ECC OS task. An example, Listing 7.15 shows such a mapping for a set of n micro-LET tasks $\lambda_{i,0} \ldots \lambda_{i,n}$, which execute the basic blocks $\tau_{i,0} \ldots \tau_{i,n}$ inside of a single OS task τ_i. With the start of the common period, the OS port uses *ILM_OS_ActivateTask* to put τ_i into its ready state. Therefore, the execution of *Call_t_i_0* starts, when τ_i is scheduled according to its priority. When *Call_t_i_0* finishes its execution, τ_i stalls due to the call of *WaitEvent*. Internally the τ_i enters the wait state, which is left when a call to *ILM_OS_SetEvent* with the corresponding event *Event_t_i* is issued. This is repeated until the execution of *Call_t_i_n* finishes. This way only one OS task is used for a set of micro-LET tasks which share the same period but use different offsets inside this period.

The previous mapping solutions have different drawbacks which might lead to a problem. For the direct mapping to BCC tasks, the maximum number of OS tasks might be an upper limit, as this is fixed due to runtime complexity and memory overhead. The grouped mapping to ECC tasks on the other hand does not have this problem. But in general, ECC tasks are to be avoided in automotive setups, since the context switch overhead increase rapidly due to additional state checking. We therefore propose a third mapping solution which maps micro-LET tasks with common periods to a single BCC task. An example for this solution is shown in Listing 7.16. General idea is to use a block sequencing based on a lookup table (*LUT_t_i[]*) and a counter (*CNT_t_i*) as table index. Each time the OS task is called, another basic block is executed based on the current counter value. Due to the increment and a modulo operation based on the lookup table size (*N_t_i*), the created block sequence is repeated for each period. The different offsets during a period are represented by the position inside of *LUT_t_i[]*. For comparison both Listing 7.15 and Listing 7.16, implement the same sequence of basic blocks. As mentioned in the beginning of this chapter, lookup based jump tables are a well-known technique and often used. It is therefore not surprising, that most modern CPU architecture provide single-cycle operations to load a jump address from a table directly to the processors program counter register. This results in a very low overhead of only a few processor cycles to load the jump address and increment the counter value.

7.2.3 Memory usage

In contrast to a system without LET, additional memory is needed for control structures and multiple value buffers. If we apply a simple double buffering, the double amount of memory is needed for global variables. Additionally, double buffer control structs with read and write pointers for indirection are needed. As mentioned before, we assume that a publisher subscriber paradigm is used and therefore values are only written by a single task. The global data written by

Listing 7.17: Memory setup on publisher

```
/* Output of T10 on C0 */
typedef struct {
  uint32 a;
  /*
   * Contains data which is
   * written by T10 on C0
   */
  uint32 y;
  uint32 z;
} OUT_C0_T10_T;


/* Buffer definition for T10 on C0 */
OUT_C0_T10_T OUT_C0_T10[2];


/*
 * Read/Write pointer control struct
 * initialized with buffer entries
 */
ILM_DATA_OUT_T OUT_C0_T10_CS = {
  .Read  = (void *)&OUT_C0_T10[0];
  .Write = (void *)&OUT_C0_T10[1];
};
```

Listing 7.18: Memory setup on subscriber

```
/* Task local pointer backup for T20 on C1*/
typedef struct {
  OUT_C0_T10_T * input_C0_T10;
  /*...*/
}IN_C1_T20_T;
IN_C1_T20_T IN_C1_T20_local;
/* Pointer backup list for T20 on C1*/
const ILM_DATA_IN_T IN_C1_T20_L[] = {
  { .pIn = &OUT_C0_T10_CS,
    .pLC = &IN_C1_T20_local.input_C0_T10 },
  /*...*/
};
/* Task local pointer backup for T20 on C2*/
typedef struct {
  OUT_C0_T10_T * input_C0_T10;
  /*...*/
}IN_C2_T20_T;
IN_C2_T20_T IN_C2_T20_local;
/* Pointer backup list for T20 on C2*/
const ILM_DATA_IN_T IN_C2_T20_L[] = {
  { .pIn = &OUT_C0_T10_CS,
    .pLC = &IN_C2_T20_local.input_C0_T10 },
  /*...*/
};
```

a single task can therefore be packed with a struct. Such an example is shown in Listing 7.17, where *OUT_C0_T10_T* represents a struct containing the global data written by τ_{10} on C_0. An array of this type with two entries (*OUT_C0_T10[2]*) then acts as the actual double buffer for all global values written by τ_{10}. In order to perform the needed indirection for read and write accesses, an additional control struct *OUT_C0_T10_CS* is used. At the end of the corresponding LET, *OUT_C0_T10_CS.Read* and *OUT_C0_T10_CS.Write* get swapped. Therefore, the payload field in Listing 7.9 references structs of type *ILM_DATA_OUT_T* for each *ILM_EVENT_PSWAP*.

Inside each subscribing task, a struct is needed to store read pointers of all relevant publishing tasks. In Listing 7.18 the corresponding variables for pointer backups are *IN_C1_T20_local* on C_1 and *IN_C2_T20_local* on C_2. In order to know which pointer should be saved during LET task activation, the ILM provides a list struct, called *ILM_DATA_IN_T*, used during LET event handling. Inside the struct a reference to the corresponding publisher output (*.pIN*) and the local backup (*.pLC*) is stored for each input. For C_1 and C_2 the resulting lists are given as *IN_C1_T20_0_L* and *IN_C2_T20_0_L* in Listing 7.18. References to those lists are then stored in the payload fields of the core local LET event table for each *ILM_EVENT_TACT* event.

Figure 7.8 shows the resulting mapping of the previously described management variables and buffers. The pointer control struct and the corresponding buffer must be located in a globally available memory, since it is accessed by publisher and subscribers. Additionally, the control struct is modified by the LET event handler of C_0. The local read pointer backup and the copy lists, processed by the cores of the subscribing tasks, can be located close coupled to the corresponding cores. Read or write accesses to either the backup stucts or the copy

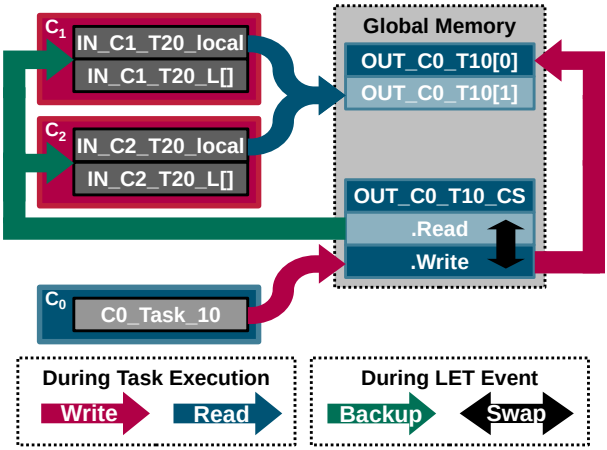


Figure 7.8: Memory location of double buffer and control structs

lists are only performed by the corresponding tasks or the LET event handling located on the core.

Comparing the local backup with the components located in the global memory shows an important difference regarding CPUs with caches. Even though the pointer backup and copy lists can also be stored in a global memory, a mechanism preserving cache coherency in not necessarily needed since the variables are only accessed by one core. In case of the control structs and buffers located in the global memory this is different. Without a coherency mechanism it could not be ensured that the subscribed tasks always read the correct values if caching is used by the CPUs. As a result, the cores of the subscribing tasks would need to flush their caches during each task activation. It is therefore highly recommended to use caching only, if the CPU implements a coherency mechanism. Otherwise, a performance loss is the case.

If we again take a look at the most prevalent AURIX μC family we realize, that the previous statement regarding the cache coherency does not hold. We already mentioned in Section 2.1.1 that TriCore CPU architecture used by the AURIX does not implement any kind of coherency mechanism. Therefore, the global memory implemented with the AURIX's LMU can not be used to store pointer control structs and buffers. Since the core local scratch pad RAMs (DSPR) can be configured to be accessible from remote cores, we recommend using those instead. In order to provide fast access for publishing tasks, pointer control structs and buffers should always be located in the DSPR of the core, executing the corresponding publishing task.

"We were first-class troublemakers. We did unconventional things in unconventional ways and still got valuable results. Thus management had to tolerate us and let us alone a lot of the time."

- Richard Wesley Hamming

CHAPTER 8

Evaluation

In order to prove the proposed mechanisms, we evaluate both approaches based on exemplary implementations, executed on existing hardware as a part of well-known RTOSs. Primary focus when designing scheduler or OS modifications is the introduced runtime and memory overhead. An evaluation regarding overhead is therefore self-evidently. Since the proposed mechanism has been developed according to different requirements, it makes sense to evaluate different aspects. In case of the proposed SPS based budget scheduling, the sufficient temporal isolation with improved response times is the most important feature. Therefore, we evaluate actual task response times for the different scheduling mechanisms and compare them to previous results of an actual RTA. This differs for the evaluation of the LET implementation. Since the actual scheduling stays the same and is independent of the LET implementation, it doesn't make sense to include this. Instead, evaluating the resulting sensitivity to additional higher priority load for an LET based system provides more interesting results since it provides an overview of the applicability of the LET paradigm. The evaluation results presented in this chapter are based on [23, 24, 25, 26, 27, 28].

8.1 SPS based budget scheduling

The analysis of the SPS based budget scheduling consists of two major parts. First an evaluation of the RTA and second, an evaluation of measured response times from an implementation. As an execution framework for the RTA pyCPA in combination with the PyPy interpreter and a multi-threaded execution on 4 cores Intel Xeon E5645@2.40GHz is used. The implementation of the proposed mechanisms are executed on an ARM926EJ-S based development board with an NXP LPC3250 on top of the existing hypervisor μC/OS-MMU.

8.1.1 WCRT Analysis

In order to evaluate response time of the SPS based budget scheduling, a set of synthetic task-sets is used and compared against the calculated response times of a TDMA based scheduling. The evaluation is based on the pyCPA framework and therefore mainly written in Python. For task-set generation we used a script from Paul Emberson [42], which is based on Roger Staffords *Random Vectors with fixed Sum* algorithm [100] and conveniently available as a Python implementation. The provided script generates a set of periodic tasks for a given utilization, with randomly distributed periods over a defined range. In order to achieve a more arbitrary task behavior, an additional random jitter, minimum distance and relative deadline is generated for each task. The priorities are assigned based on RMS, according to the task periods.

For the analysis evaluation two different scenarios are used, both with an overall number of four partitions, where one partition is reserved for the hypervisor itself. In a real implementation, this reserved partition is often used for housekeeping and can be characterized for the analysis with a single task. For evaluation, the utilization of the hypervisor partition is fixed to 4%. The three remaining *application partitions* got each a synthetic task-set assigned with a random utilization between $15\% \ldots 20\%$, generated by the previously mentioned script from Emberson et al. For each task inside a partition an activation jitter was assigned randomly limited to 50% of the tasks period. Same was done for deadlines, which are defined randomly at generation between $75\% \ldots 125\%$ of the corresponding tasks period.

For the first scenario, 60 setups has been generated with a varying number of tasks from two to four inside the application partitions. Figure 8.1 shows the steps, performed during evaluation. For each setup the optimization algorithm from Section 4.5.1 is executed with a step granularity of $\Delta = 1\mu s$. The calculated slack is then distributed equally based on (4.28). The optimization provides a set of possible TDMA configurations for the hypervisor scheduling. From this set all configurations with 0%, 5%, 10%, 15% or 20% slack are used for further evaluation.

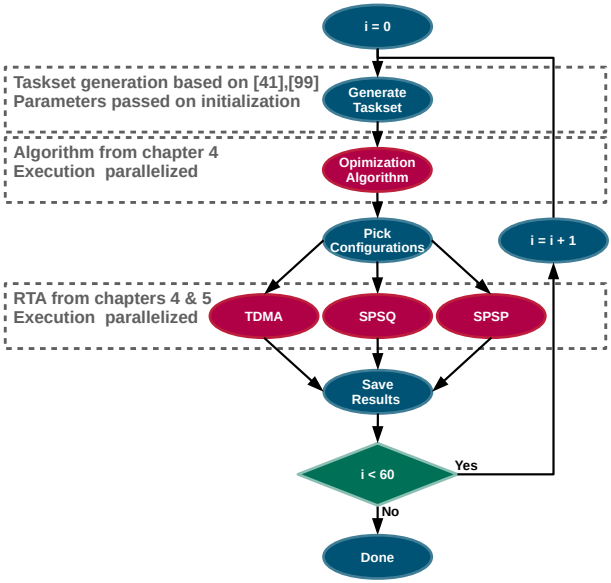


Figure 8.1: Evaluation flowchart

Next, for each task in each configuration three different RTAs are performed. The first analysis is based on the standard TDMA busy-window from (4.13). Without the usage of any background scheduling, the calculated WCRT for an SPS based system are equal to the TDMA values, if the SPS configuration has been derived from a TDMA configuration as described in Chapter 5. This is obvious when comparing (5.5)/(5.6) to (4.13)/(4.17). Same for the mentioned IRQ shaping modification from Section 4.3. Both scheduler modifications only influence the average case performance of either IRQs or tasks. Since this section only considers the formal worst case, WCRTs are not calculated for both mentioned modifications since those are equal to the standard TDMA RTA. Instead, the WCRTs for an SPS based system with queue based background scheduling (SPSQ) (according to the conservative equation (5.19)) and priority based background scheduling (SPSP) (according to the extensive method proposed in Section 5.2.2) are calculated. This is repeated for all tasks in the system and all picked configurations. In the end this results in $\sim$ 870000 configurations from 60 different task setups, analyzed with three different methods.

The results for the first scenario are shown in Figure 8.2. Because of the varying number of tasks in a system setup, we tried to provide a compact representation based on box plots. The boxes denote the range between 25% $\rightarrow$ 75%,

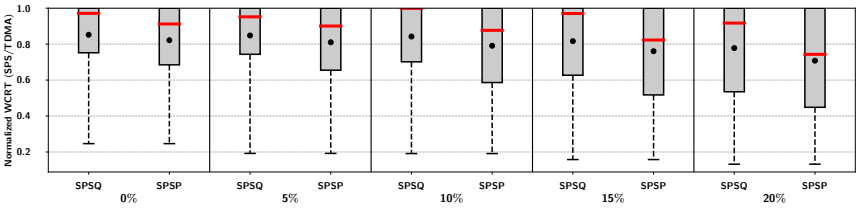


Figure 8.2: Merged representation of calculated SPS based WCRTs relative to TDMA

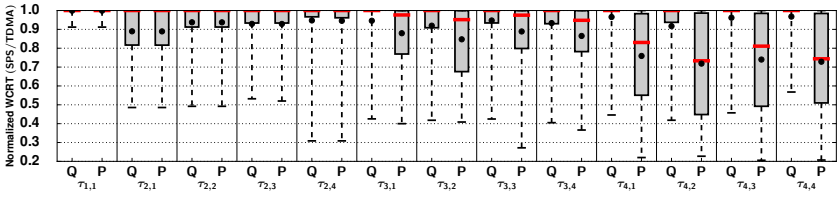
the black dot indicates the average and the red bar the median of all calculated values. In general for each task WCRT a normalized value relative to the corresponding TDMA based WCRT for the same configuration is calculated. A value equal 1.0 in this plot means, that the SPS based budged scheduling for the considered task and the current configuration does not provide a better WCRT compared to TDMA. Values < 1.0 therefore represent an improvement compared to TDMA. This way it is possible to compare the calculated WCRTs for all task in a single graph, even though the number of tasks may change in different configurations. As an example, the box plot Q from SPS_0 contains the WCRTs of all generated task under SPS based budget scheduling with a background scheduling implementing queues for all evaluated configurations with 0% slack in the corresponding TDMA schedule. In contrast to this, P from SPS_0 contains the WCRTs from the exact same configurations with 0% slack, but with a priority based background scheduling, assinging a disting background priority to each partition. The collected results show that for all calculated WCRTs both SPS based methods with background scheduling provide at least the same WCRT bound, but often a way better value compared to TDMA. None of the calculated WCRTs for both methods were greater than the corresponding TDMA WCRT considering the same configuration.

As you can see, it is more likely to achieve better WCRTs compared to the standard TDMA scheduling if the slack inside the system increases. Also, prioritizing single partitions during background scheduling may also improve the WCRTs of the considered partition. When considering the worst-case behavior, this does not have any effect to the WCRTs of the remaining partitions with smaller priorities. Reason for this is, that the analysis of a queue based background scheduling always assumes the maxmimum blocking from other partitions, which is equal to the lowest priority in a priority based scheduling. In case of actual measured response times, this is different and will be shown in Section 8.1.2. The results also show that for some tasks and some configurations a SPS based system does

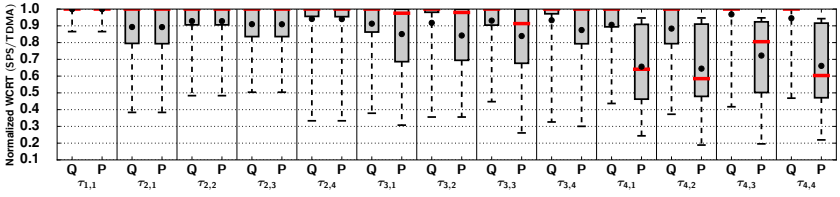
not provide better WCRTs, since all box plots reach up to 1.0. This is due to the fact that the box plots include the results for all tasks of each generated setup. Therefore, also tasks are included which would never see any benefit, based on its task parameters e.g. based on its partition internal priority or its period.

In order to explain this further more, a second evaluation is performed. Again a parameter setup with 60 task-sets as well as identical generation parameters for utilization, period, priorities, jitter and deadlines is used. Also, a distinct hypervisor partition is included in each task-set, leaving room for three additional application partitions. Again the same evaluation flow as described in Figure 8.1 is used. The major difference is that the number of possible tasks is fixed to four for each application task-set. This way it is possible to show the different calculated response times for each task individually. Like before only a subset of all possible configurations with either 0%, 5%, 10%, 15% or 20% slack are taken for further calculations leading to $\sim$ 990000 configurations. Again, for each task in each configuration a RTA is performed. The results for queue based and priority based background scheduling relative to TDMA are shown in Figure 8.3. Every task in the system is listed individually and therefore the influence of a priority based background scheduling can be seen better compared to Figure 8.2. In general the tasks are described as $\tau_{p,i}$ with p indicating the partition and i a task inside this partition. The priority inside partitions is ordered in reverse, resulting in $i = 1$ as highest task priority. For the priority based background scheduling, we define partition $p = 4$ to be the highest priority and repeat this for all remaining partitions in descending order. This results in the lowest background priority for the hypervisor partition $p = 1$ and its single task $\tau_{1,1}$.

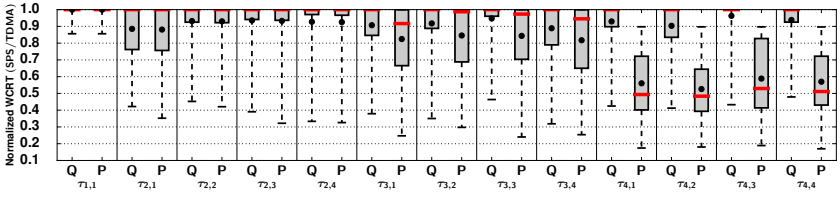
The WCRT distribution is shown for all settings from 0% up to 20% slack in Figure 8.3a…8.3e. First, a task with a high priority inside a partition might see much lesser improvement, especially for configurations where the amount of slack is small. Important is at this point the amount of background scheduling included during the busy-window of the task under analysis. A high priority task inside a partition is scheduled first, when the corresponding partition is dispatched by the hypervisor. This means, that the busy-window of such a task might not contain any background scheduling, resulting in the same WCRT compared to TDMA. In contrast to this, a lower priority task inside a partition generally sees more interference due higher priority inside the partition. This leads to a longer busy-window which therefore might contain background scheduling. Second, the benefit of a priority based background scheduling can be seen for tasks inside partitions 4 and 3. Comparing queue based (Q) and priority based (P) background scheduling shows a significant improvement for tasks in partitions with a higher background priority. This is especially the case for configurations with more slack (e.g. Figure 8.3e and 8.3d). Here the tasks inside a partition ($\tau_{4,1} \ldots \tau_{4,4}$) with a high background priority achieve significant better calculated



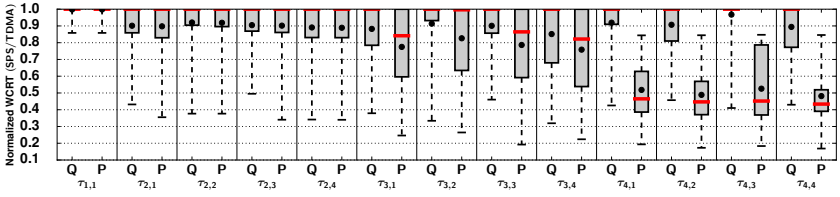
(a) Distribution for 0% slack



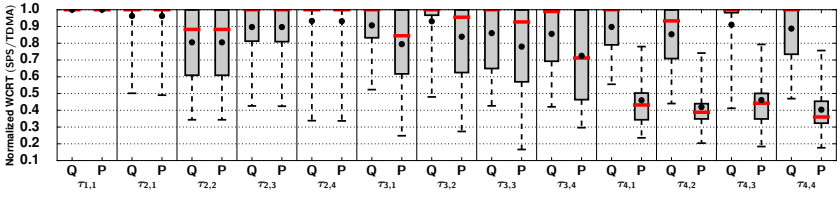
(b) Distribution for 5% slack



(c) Distribution for 10% slack



(d) Distribution for 15% slack



(e) Distribution for 20% slack

Figure 8.3: Comparison of queue and priority based background scheduling

WCRTs for all tested configurations with 20% slack.

The two previously described scenarios resulted in ~ 1860000 different configurations. Evaluating the WCRTs for all tasks of those configurations took ~ 7 hours on an Intel Xeon E5645@2.40GHz. In general, analyzing the runtime of fixed-point iterations is always a challenge. The busy-window itself must be iterated several times until a fix-point is reached. In case of the multiple event busy-windows the entire iteration might be repeated several times for a number of events. Both, the number of iteration and the number of considered events, heavily depend on the task-set. Since all the proposed mechanisms are constructed based on the busy-window technique, this dependency can be seen in all methods. The cycle optimization is basically a busy-window based TDMA RTA performed several times with different parameters. The number of RTAs to be performed is directly related to the theoretical upper bound from (4.27) and the used stepsize Δ.

Since all values are identical in case of the described SPS configuration, the RTA of an SPS based system without background scheduling does not only provide the same WCRT, it also takes the same amount of time. The RTA of a system with queue based background scheduling based on (5.19) also results in a more or less equal runtime complexity compared to a TDMA RTA. At least for the complex calculation method from Section 5.2.2, a comparison is possible. One might see, that based on the different orders from (5.17), the entire busy-window iteration is executed for each possible partition order. Therefore, the analysis runtime for such a system scales based on the number of possible partition orders Y, which is given as:

$$Y = (|\Gamma_{HYP}| - 1)! \tag{8.1}$$

Due to this fact, the complex calculation method takes much more time for computation. Nevertheless, only the complex analysis method for SPS based systems is able to handle different types of background scheduling. Instead, the simplified method from (5.19) can only handle queue based systems. It is still a design-time problem, which is not relevant during runtime. Also, calculating response times for a single configuration is comparatively fast on modern computers. The following numbers in Table 8.1 are used to highlight this and show the actual analysis runtime for a single task-set from Table 8.2, which is later used in Section 8.1.2 for actual runtime measurements. The optimization is performed on all possible time cycle sizes with a stepsize of $1\mu s$. The actual RTAs are then performed only on a single configuration with the maximum amount of absolute slack inside the TDMA schedule. All measurements include the overhead of the used pyCPA implementation.

Optimization	$\sim 5200ms$
TDMA/SPS	$\sim 50ms$
SPSQ	$\sim 55ms$
SPSP	$\sim 65ms$

Table 8.1: Analysis runtime in [ms]

8.1.2 Response time measurements

For evaluation, all proposed mechanisms are tested in an implementation on top of an existing hypervisor architecture. This includes the proposed IRQ shaping as well as the SPS based budget scheduling with and without background scheduling (queue and priority based). The modified hypervisor setup is based on μC/OS-MMU [41] with μC/OS-II [68] as guest OS inside partitions. While μC/OS-II does not directly implement an OSEK or POSIX API, it provides a simple SPP scheduling with up to 256 priorities, where 0 denotes the highest priority in the system (or partition in this case). In order to implement the $CB\_Idle(p)$ callback, a service call to the hypervisor is added inside the partitions via a hook function of the μC/OS-II idle-task. This way the service call is always invoked when a partition has nothing to do, which is exactly the behavior explained in Section 7.1.3. Also, the $CB\_Resume(p)$ callback is invoked for each partition-level task activation as well as for IRQs to achieve the desired behavior.

The applications inside the partitions are simulated by a generic task-set implementation which allows a simple execution time simulation. The task parameters are shown in Table 8.1 and represent a simple PJ-model with period and jitter. Each task activation is generated with a timer according to the corresponding period P and pseudo random jitter within the range of J. The execution time $\overline{C}$ is then simulated with a processor specific inline assembly. If a task misses its deadline D, a service call to the hypervisor core is generated and saved for later evaluation. The evaluation setup from Table 8.2 again consists out of four partitions, where one partition is reserved for the hypervisor itself. Again this leaves room for three application partitions with four tasks each. Since the hypervisor partition is used for housekeeping like internal garbage collection or serial debug output, the assigned single task in Table 8.1 is only artificial and used during the time cycle optimization. Speaking of this, the time cycle optimization results in a timecylce length T_{TDMA} and timescycle slack T_S. All times are given in milliseconds.

$$T_{TDMA} = 48.3, \qquad T_S = 14.3$$

Distributing the slack equally according to (4.28) and adding this to the minimal timeslot sizes $T_{1,min} \ldots T_{4,min}$ leads to:

Hypervisor ($T_1 = 2.8$)				
Prio	**P**	**J**	**$\overline{C}$**	**D**
1	100	0	4	100

Partition 1 ($T_2 = 11.4$)				
Prio	**P**	**J**	**$\overline{C}$**	**D**
1	50	5	2	50
2	100	5	4	100
3	200	5	6	100
4	400	5	10	200

Partition 2 ($T_3 = 18.0$)				
Prio	**P**	**J**	**$\overline{C}$**	**D**
1	50	5	3	50
2	75	5	6	75
3	150	5	7	150
4	175	5	10	175

Partition 3 ($T_4 = 16.1$)				
Prio	**P**	**J**	**$\overline{C}$**	**D**
1	100	5	4	75
2	150	5	6	85
3	200	5	8	150
4	250	5	12	175

Table 8.2: Evaluation task-set, all times are given in [ms]

$$
\begin{aligned}
T_{1,min} &= 2.0 & T_{2,min} &= 8.0 & T_{3,min} &= 12.7 & T_{4,min} &= 11.3 & \\
T_{1,S} &= 0.8 & T_{2,S} &= 3.4 & T_{3,S} &= 5.3 & T_{4,S} &= 4.8 & \\
T_1 &= 2.8 & T_2 &= 11.4 & T_3 &= 18.0 & T_4 &= 16.1 & \\
T_{TDMA} &= T_1 & &+ T_3 & &+ T_3 & &+ T_4 & = 48.3
\end{aligned}
$$

In order to evaluate the proposed runtime mechanisms, the task-set from Table 8.2 is executed based on the previously derived configuration for the following hypervisor setups:

1. **TDMA**: Default TDMA scheduling used as baseline
2. **TDMAS**: TDMA + IRQ shaping
3. **SPS**: SPS w.o. background scheduling
4. **SPSQ**: SPS w. queue based background scheduling
5. **SPSP**: SPS w. priority based background scheduling

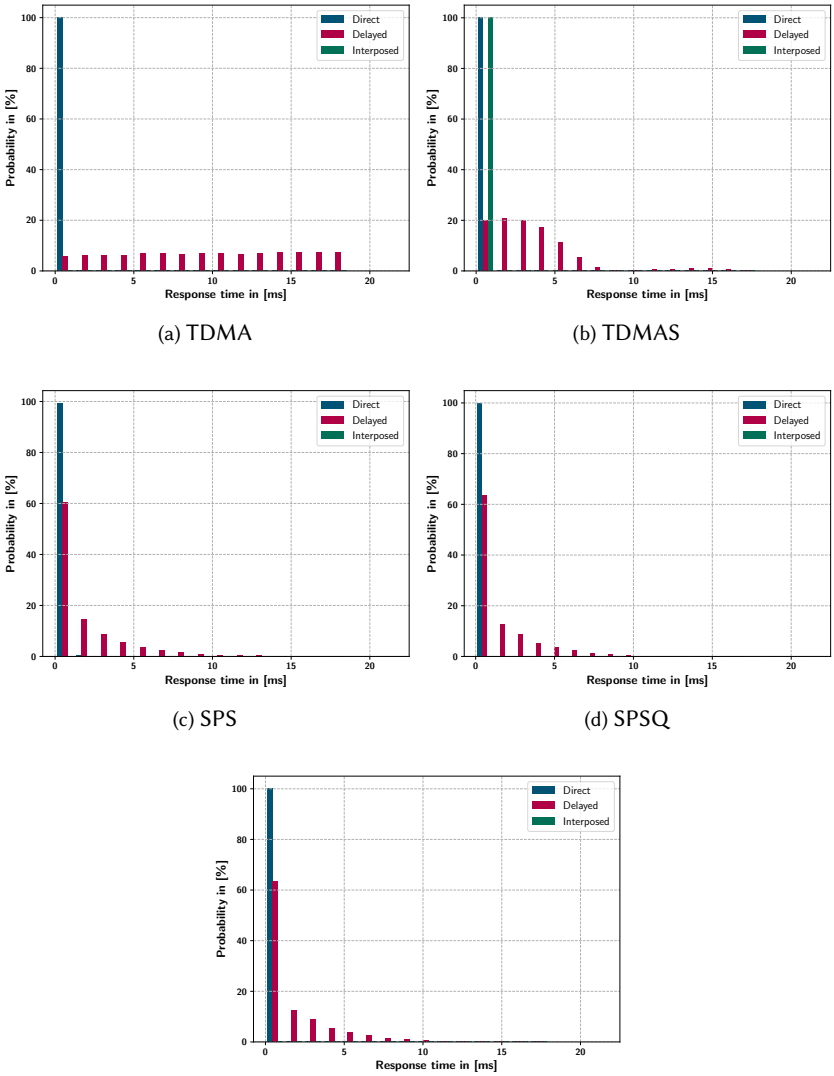
Since especially the IRQ shaping, but also the SPS based budget scheduling, target to improve the response times of IRQs, those need to be evaluated as well. In order to test this, an exponentially distributed IRQ pattern, generated by one of the processor's hardware timers, is used. The distributed IRQ pattern is characterized by the mean interarrival rate given as:

$$\lambda = U / \overline{C}_{BH} \tag{8.2}$$

For each system setup 14000 IRQs with a mean load $U = 30\%$ and $\overline{C}_{BH} = 500\mu s$ are generated. The appropriate BH is located in partition 1 (T_2). Therefore, the monitors of the remaining partitions during the TDMAS test get configured according to Section 4.3. All other tests use the exact same IRQ pattern but without any IRQ shaping, therefore no monitoring needs to be configured for those tests.

Before we focus on the actual response times, we first consider the different IRQ handling types. As a kind reminder of Chapter 4, an IRQ is markt as *Direct* handled if it occurs during execution of the corresponding partition. If an IRQ occurs during the execution of a foreign partition, it is stored and handled afterward when the corresponding partition is executed again. In that case an IRQ is marked as *Delayed*. The IRQ shaping provides a third option, which handles an IRQ *Interposed* and preempts the currently scheduled partition. Figure 8.4 shows the distribution of this different IRQ handling methods as histograms over the measured response time. To clarify this, Figure 8.4a show that in case of a standard TDMA scheduling all *Direct* IRQs have a short response time. In contrast to this, the response times of *Delayed* IRQs are more or less equally distributed across a wider response time range. When using IRQ shaping this distribution changes, as shown in Figure 8.4b. All *Interposed* IRQs get handled immediately instead of *Delayed* and show therefore a short response time in Figure 8.4b. As the absolute number of delayed IRQs decreases, the relative probability for *Delayed* IRQs shifts towards the left.

For the SPS based scheduling mechanisms the results are shown in Figure 8.4c, 8.4d and 8.4e. Even though an *Interposed* IRQ handling is not available for this scheduling mechanisms, it performs similar, since even *Delayed* IRQs get mostly handled within a short time. Primary reason for this is the SPS based mechanisms, which adapts the budget provisioning to the actual usage. Compared to a standard TDMA scheduling this leads to an adaptive usage of the available budget and therefore shorter response times for *Delayed* handled IRQs.



(a) TDMA

(b) TDMAS

(c) SPS

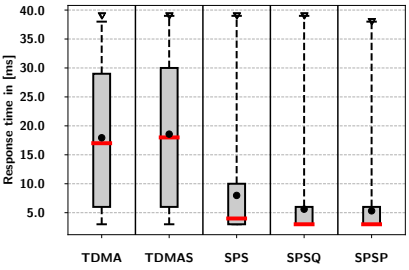
(d) SPSQ

(e) SPSP

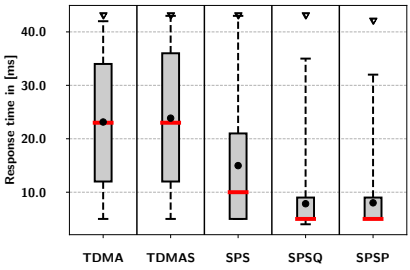
Figure 8.4: Response time distribution for different IRQ handling paths

Next we take a look at the actual response times of both IRQs and tasks, collected for the different scheduling techniques. During all tests no task deadline has been violated. As already mentioned in Section 7.1.3, the provided implementation is based on multiple queues and the insertion method defines the actual background scheduling. In order to provide measurements also for priority based background scheduling, we prioritized *Partition 1* over all other partitions. Inside the implementation this is a simple task, as the queues are based on double linked lists. Therefore, adding a partition at the head instead of the tail corresponds to adding it with the "highest" priority to the background scheduling. The results presented in Figure 8.5 and 8.6 only show the measurements from *Partition 1* in order to highlight the difference between queue based and priority based background scheduling.

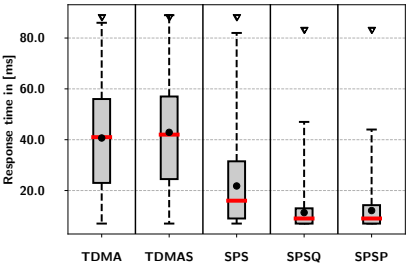
The results in Figure 8.5 show the measured response times as box plots. Like before, the boxes denote the range between 25% $\rightarrow$ 75%, the black dot indicates the average and the red bar the median of all measured values. Additionally, the ∇ symbols indicate the calculated WCRTs. Comparing the standard TDMA scheduling to a scheduling with IRQ shaping (TDMAS) shows for all tasks, that the improvement of IRQs is achieved at the expense of tasks. Even though the SPS without any background scheduling provides the same worst-case, the average case improvement compared to TDMA can directly be seen for both IRQs and tasks. The more flexible SPS scheduling provides much better response times to higher priority tasks or IRQs which get processed right after a partition was dispatched. With background scheduling, this is improved further more also for lower task priorities. While the worst-case for tasks with higher priority inside the partition is more or less the same compared to TDMA for both background scheduling techniques, it significantly improves for lower task priorities. This is due to the already explained reason that higher priority tasks may not benefit from background scheduling during the RTA. Regarding the actual measurements, an improvement is clearly visible, even though the difference between SPSQ and SPSP is only minor for the used task-set. In order to show this improvement further more, Figure 8.6 shows the response time histograms for the four considered tasks of *Partition 1* and the corresponding IRQ. Again a shift towards the shorter response times is clearly visible for the SPS based scheduling mechanisms. Important is at this point, that this is achieved while preserving a comparable improvement like TDMAS also for IRQs compared to standard TDMA. Also, keep in mind that the mechanisms providing this improvement, still ensures a sufficient temporal isolation according to Definition 4.3.



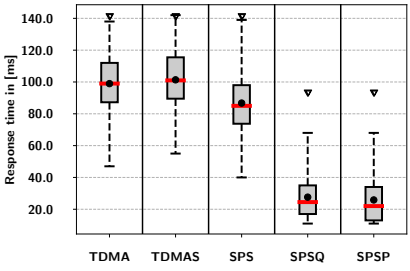
(a) Task $\tau_{2,1}$



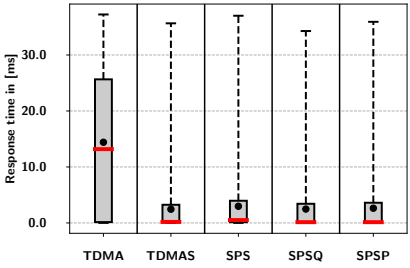
(b) Task $\tau_{2,2}$



(c) Task $\tau_{2,3}$

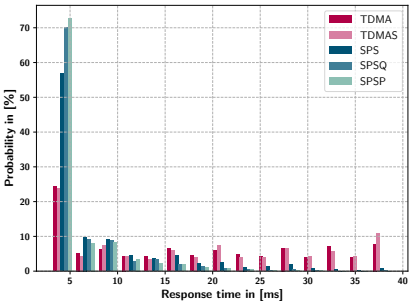


(d) Task $\tau_{2,4}$

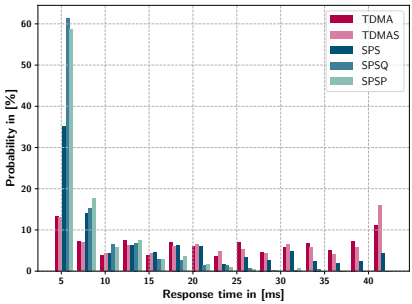


(e) IRQ

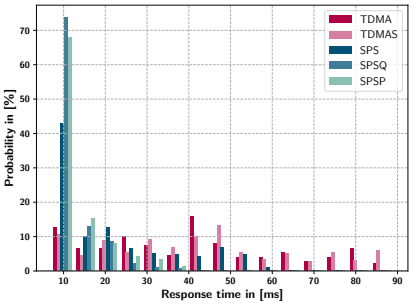
Figure 8.5: Response time measurements for TDMA and SPS based scheduling shown as box plots



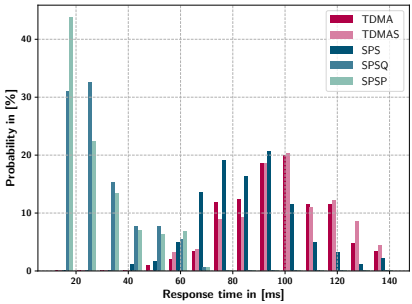
(a) Task $\tau_{2,1}$



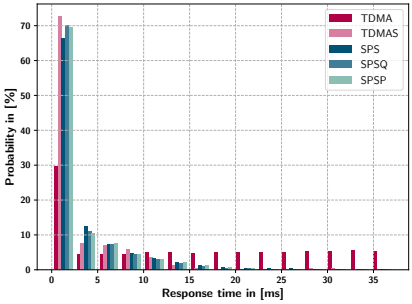
(b) Task $\tau_{2,2}$



(c) Task $\tau_{2,3}$



(d) Task $\tau_{2,4}$



(e) IRQ

Figure 8.6: Response time measurements for TDMA and SPS based scheduling shown as histograms

8.1.3 Runtime and memory overhead

Table 8.3a shows a comparison of the hypervisors memory usage for three different scheduler configurations. First the original TDMA scheduler, second the TDMA scheduler including the IRQ shaping mechanism and third the SPS based mechanisms with budget scheduling. The original TDMA scheduler is in all areas the smallest one. But keep in mind that here a commercial implementation is compared with two academic implementations, which might have not been optimized to the limit. While TDMAS only required a modification of the existing scheduler, is has been replaced entirely within the SPS implementation. Only the actual dispatch mechanisms have been recycled. Both modified versions show a similar code size and memory usage. An important thing to mention is that the monitoring used for the IRQ shaping does not have a constant overhead for different scheduling parameters. When changing the monitors trace length (stored in the data segment) as mentioned in [89], this does affect memory usage as well as the static runtime overhead. As described in Section 4.3, the trace length depends on the actual amount of available slack. Therefore, the overhead would increase for systems with more slack. As the SPS does not use this mechanism anymore, the memory usage and the runtime overhead are independent of the schedulers parameters.

Table 8.3b shows the WCETs of the most important parts of the SPS implementation with background scheduling. As mentioned before, background scheduling is based on an additional queue with only a lazy implementation of a single additional background priority for evaluation. The gathered values therefore present the SPSQ case, which is more or less identical to the used implementation during the SPSP evaluation. Since measuring WCETs is a fairly complex task, we used OTAWA [20] to derive this values based on a static assembly level code analysis. Without any runtime dependent loops inside the code and statements primarily based on *switch case* resulting in look up tables, this should provide a sufficient estimation. The LPC3250 on the used development board is clocked at

	TDMA	TDMAS	SPS Q/P
Code	121964	124808	125548
Data	381576	382576	381776
Sum	**503540**	**507384**	**507324**

(a) Memory usage in bytes

	CPU Cycles	
	Handler	Callback
Refill	3840	1555
Empty	3445	2690
Resume	4760	2345
Idle	4885	1255

(b) WCET in cycles

Table 8.3: System overhead

200MHz which results in $200\frac{Cycles}{\mu s}$. Therefore, 2000 cycles correspond to $10\mu s$. While the proposed mechanisms can be implemented with limited memory overhead as mentioned before, the actual execution time overhead still depends on the functions of the underlying OS or RTE. As an example, the scheduler callback functions use the underlying queue system which itself is based on the internal memory management of μC/OS-MMU. In order to push an entry to one of the queues, first the memory of the entry is allocated based on the underlying memory management. This step on its own may take up to 1000 cycles, leaving lots of room for improvement.

8.2 LET implementation with Zero-Time Communication

This section evaluates the proposed Zero-Time Communication based on the ILM implementation. First an overview of the memory and runtime overhead is given and second the behavior under overload is considered. The evaluation is based on an example from Daimler RD/EIS, which represents a micro-LET based distribution of application software from an ECU of the powertrain subdomain. As execution platform an Infineon AURIX TC275TF@200MHz and ERIKA OS 2.7 is used. The corresponding hardware and OS port are part of the freely available ILM [7]. The entire system is compiled and optimized for speed with a gcc 4.6.4 based toolchain (gcc -O2). All runtime values have been measured with a Lauterbach Trace32 setup, including a 17-channel logic sniffer for calibration measurements via pin toggling. The evaluation is performed for the following three task mapping mechanisms, which already have been discussed in Section 6.3.

BCC: Direct mapping to BCC

BCC+: Grouped mapping to BCC

ECC: Grouped mapping to ECC

8.2.1 Runtime and memory overhead

As discussed in Section 7.2, each LET event is connected to an IRQ generated either by the underlying timer hardware or through a redirected software IRQ. Resulting from this, the discussed activation delay from Section 6.3 is given as distance between the IRQs occurrence and the start of execution within the task's context. Table 8.4 shows the measured overhead for the three proposed integration methods and Figure 8.7 visualizes the temporal sequence.

After occurrence of an IRQ, the hardware specific low-level handler is called. This is shown in Figure 8.7 and Table 8.4 as $B_{R\&R}$. In case of the used AURIX μC, $B_{R\&R}$ is the time needed to reset IRQ flags and reconfigure the CCU6 to

	BCC	BCC+	ECC
$B_{R\&R}$	0.9		
B_{C2CI}	0.6		
B_{LET}	1.4		1.5
B_{C2CS}	1.8 (0.9)		
B_{CTX}	1		2
B_{FBC}	0	0.1	0
B_{ACT}	**5.7 (4.8)**	**5.8 (4.9)**	**6.8 (5.9)**

Table 8.4: Measured activation delay for different implementations in μs

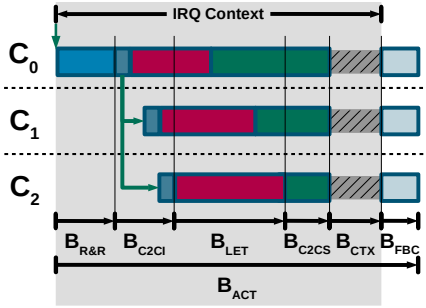


Figure 8.7: Activation delay

the next compare value. Since the steps taken for the BCC, BCC+ and ECC are always the same, the execution time does not differ between the different mapping mechanisms. After the reconfiguration of the underlying timer hardware, a software IRQ is forwarded to the slave cores. The time needed to forward and process this IRQ is given as B_{C2CI} and equal for all three mapping mechanisms. After the software IRQs have been redirected or handled, the LET event handling is executed. Measuring the runtime overhead for the processed LET events is crucial. Reason for this is the fact, that each LET IRQ might trigger multiple LET events. A simple example is the back-to-back execution of two tasks. In this case the first event swaps the buffer of the previous task and the second event activates the subsequent task. The LET handling overhead B_{LET} shown in Table 8.4 is exactly for this scenario. In case of an ECC based system, the overhead is slightly larger. An ECC based system might use the *ILM_OS_SetEvent* call, which are more expensive than an additional call of *ILM_OS_ActivateTask*. After the event handling was executed, each involved core enters a synchronization barrier. The synchronization barrier is exited by all cores at the same time. This means, that the core which has received the worst-case blocking yet and enters the barrier last, only checks it once. In Figure 8.7 this is visualized for C_2 with B_{C2CS} representing the overhead for checking the synchronization barrier once. Again this part is independent of the used conformance class. We measured two different values for B_{C2CS}, as we realized a comparatively large overhead of $1,8\mu s$ for the default synchronization barrier implementation of ERIKA OS. Therefore, we stripped this implementation and removed a not necessary spinlock call, resulting in only $0,9\mu s$ overhead. The context switch overhead B_{CTX} differs as expected for both conformance classes. While the low-level context switch itself does not take longer in case of an ECC based system, the additional event handling *WaitEvent/ClearEvent* is responsible for the additional overhead (compared to both BCC setups). The overhead for the function block call B_{FBC} only

	10 ms					20 ms							100 ms		
λ	10,0	10,1	10,2	10,3	10,4	20,0	20,1	20,2	20,3	20,4	20,5	20,6	100,0	100,1	100,2
LET	1400	350	300	500	1200	700	400	600	250	400	500	1000	900	800	400
WCET C_1	628	77	129	221	594	333	-	137	125	133	353	463	423	362	185
WCET C_2	-	169	154	-	-	-	188	293	-	172	-	-	-	337	-

Table 8.5: Micro-LET evaluation use-case, times given in [μs]

occurs for a BCC+ setup with a function block sequencing inside the OS task (Listing 7.16) and is comparatively small as expected. With those values, B_{ACT} can be constructed. The values in parentheses are based on the modified barrier synchronization.

Even though Table 8.4 shows the activation delay for each task, the runtime overhead primarily depends on the actual application which should be coordinated on multiple cores by the LET paradigm. The actual number of IRQs and events depends on the LETs of the coordinated application, which therefore influences the runtime overhead for event handling. Table 8.5 therefore shows an abstracted micro-LET use-case from an automotive powertrain controller. The use-case is abstracted in a way, that only sizes of LETs and execution time bounds are provided. While the LETs values are used to generate the corresponding configuration tables, the WCET values get simulated, like in Section 8.1.2, with an architecture specific inline assembly. Overall, the use-case consists of 15 LET tasks, with corresponding OS tasks on C_1 and C_2 (called application cores), while the corresponding IRQ pattern is generated on C_0. Due to the periods of $10ms$, $20ms$ and $100ms$ the hyperperiod of the described use-case is $100ms$. Empty entries inside the WCET rows indicate, that no OS task is executed during this LET on the corresponding application core.

In order to test the proposed use-case, the available hooks of the ILM are used for pin toggling, visualized via the 17-Bit logic sniffer of a Lauterbach PowerTrace module. Figure 8.8 shows this for the BCC+ implementation of the use-case from Table 8.5. The meaning of the showed signals is:

LET_IRQ_Cx: Set to 1 when processing an LET IRQ or event. The primary IRQ is generated on C_0 with the CCU6 and redirected to C_1/C_2 via a so called General Purpose Service Request Node (GPSRN) available in the AURIX μC family. x is replaced with the corresponding core (0/1/2).

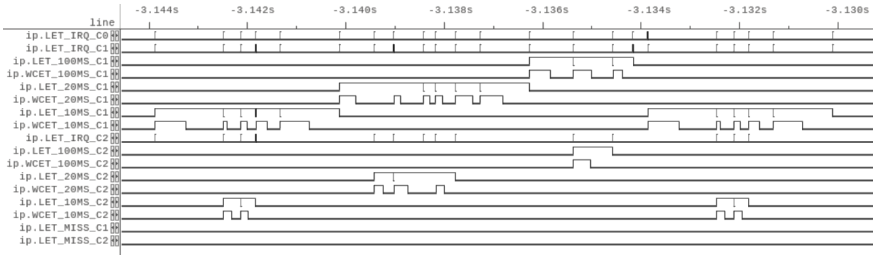


Figure 8.8: Execution trace of the use-case from Table 8.5

LET_yMS_Cx: Set to 1 if an LET task with period y (100/20/10) is activated on application core x. Set to 0 at the end of an LET task. As the LET tasks are sequenced back-to-back for a micro-LET system, only a small peak is visible between LET tasks with the same period y but different offset.

WCET_yMS_Cx: Set to 1 when the execution of the corresponding function block inside the corresponding OS task starts. Set to 0 when the block function finishes its execution. Therefore, high priority interference is indicated through a delayed execution start or an execution that takes longer than the corresponding WCET. Again y denotes the period and x the application core (1/2)

LET_MISS_Cx: Temporarily set to 1 if a block function is still executing, although the LET has elapsed. x is replaced with the corresponding application core

Figure 8.8 shows the start of a hyperperiod, where LET tasks of all three periods get activated after each other. The configuration from Table 8.5 can directly be recognized in Figure 8.8. This is possible, since the LET task sizes in the showed use-case are equal to twice the WCET, based on a design constraint. Due to lack of high priority interference, the OS tasks start their execution directly after the activation of the LET task.

In order to calculate the overall overhead introduced by the LET handling, the WCET of each handler activation ($\overline{C}_{Hdl,x}$) and the overall number of handler activations per hyperperiod are necessary. Therefore, we get for each core x:

$$U_{Hdl,x} = \frac{NoOfHandlerActivations_x}{Hyperperiod} \cdot \overline{C}_{Hdl,x} \tag{8.3}$$

The number of activations can directly be derived from the proposed LET sched-

ule given in Table 8.5. It results in 98 hardware IRQs on C_0, where all 98 IRQs are relevant for C_1 and only 58 for C_2. $\overline{C}_{Hdl,x}$ can be measured for each core during runtime with the evaluation setup, but it can also be constructed based on Figure 8.7 and Table 8.4. Important is that $\overline{C}_{Hdl,x}$ includes different parts for different cores. As an example, while $\overline{C}_{Hdl,0}$ might include overhead for the CCU6 reconfiguration (given as $B_{R\&R}$) it does not include the actual LET event handling B_{LET} as no application is executed on C_0. In contrast to this, $\overline{C}_{Hdl,1}$ and $\overline{C}_{Hdl,2}$ include LET event handling but don't reconfigure the CCU6. Nevertheless, for all cores the context switch overhead must be included and also parts of the core-to-core IRQ based on the GPSRN. This results for the BCC+ implementation in:

$$U_{Hdl,0} = \frac{98}{100000\mu s} \cdot 2.0\mu s \approx 0.20\%$$
$$U_{Hdl,1} = \frac{98}{100000\mu s} \cdot 4.5\mu s \approx 0.45\%$$
$$U_{Hdl,2} = \frac{58}{100000\mu s} \cdot 4.5\mu s \approx 0.26\%$$

Distributing application and IRQ generation differently also influences the caused overhead. Using only cores C_1 and C_2, with IRQ pattern generation on C_1, the overhead results in:

$$U_{Hdl,0} = 0\%$$
$$U_{Hdl,1} = \frac{98}{100000\mu s} \cdot 7.0\mu s \approx 0.70\%$$
$$U_{Hdl,2} = \frac{58}{100000\mu s} \cdot 4.5\mu s \approx 0.26\%$$

Important is at this point, that in this case the blocking based on the core-to-core IRQ must be considered entirely. Even though only the first part (setting the IRQ) is executed on C_1, the second part on C_2 (handling the IRQ) is considered indirectly through the synchronization barrier at the end. This is not the case, if there is no LET event handling on the core used for IRQ generation. As an additional example, moving the pattern generation to C_2 would increase the overhead unnecessarily, as now all IRQs are also handled on C_2, even though only 58 IRQs are relevant for the local application. This would result in:

$$U_{Hdl,0} = 0\%$$
$$U_{Hdl,1} = \frac{98}{100000\mu s} \cdot 4.5\mu s \approx 0.44\%$$
$$U_{Hdl,2} = \frac{98}{100000\mu s} \cdot 7.0\mu s \approx 0.70\%$$

Which distribution method is preferable, depends heavily on the rest of the system. While, combining IRQ pattern generation directly with event handling on the same core removes the need of an additional core for pattern generation, it clearly increases the overhead of the LET handling. Also, LET events that are only relevant for an application on another core do now cause additional interference to the local application. This is not the case for a setup where the IRQ pattern generation is executed on a distinct core, as only events relevant for the local application are redirected to corresponding cores.

Like the runtime overhead, also the memory overhead of the proposed ILM implementation primarily depends on the number of LET IRQs and events during a hyperperiod. The determined memory usage is shown in Table 8.6. All results are based on the BCC+ implementation where the application is mapped to C_1 and C_2. C_0 is only used for IRQ pattern generation. The meaning of the five different memory sections is:

.text: Instructions

.bss: Uninitialized RAM variables

.data: Initialized RAM variables

.rodata: Read only variables

.global: Global RAM variables for core-to-core interactions

In order to show the important parts, the results are devided into three different parts. First, the LET implementation including the relevant event handling, as well as the OS and hardware port. Second, the hook functions used for pin toggling and tracing and third the application including WCET simulator and the IRQ/event tables. The LET implementation primarily consists of instructions in .text and a few uninitialized system variables in .bss. In case of C_0 also the AURIX specific hardware initialization as well as 8 bytes for bitmasks and barrier in the .global section, implementing the core-2-core synchronization, are included. For additional tracing via a logic sniffer, the hook functions implement a pin toggling resulting in additional code overhead in .text. The overhead in .data and .rodata is based on pin mapping tables which provide an abstraction layer to the used evaluation platform.

	.text	.bss	.data	.rodata	.global
LET implementation					
C0	638	62	0	0	8
C1	200	17	0	0	0
C2	200	17	0	0	0
Hook functions					
C0	166	0	184	192	0
C1	146	0	0	184	0
C2	158	0	0	184	0
Application					
C0	0	0	0	294	0
C1	476	6	0	1724	0
C2	170	6	0	952	0

Table 8.6: Memory overhead for the use-case from Table 8.5

The simulated application also generates overhead on all cores. On C_0 the application dependent IRQ table is locate, holding 98 entries for the use-case from Table 8.5. According to Listing 7.8 each entry consists of a relative offset for the timer and a bitmask for the corresponding cores. In case of the AURIX μC this can be encoded in 3 bytes (1 byte bitmask, 2 bytes offset) leading to 294 bytes for 98 entries. As the table should never be changed during runtime, it is located in the .rodata section.

On C_1 and C_2 the application simulation framework generates some overhead in the .text and .bss section. Since the simulation relevant code is only needed due to the lack of a real application, it can be neglected. Instead, the event tables stored in .rodata are much more important. As already discussed before, the application reacts to 98 relayed IRQs on C_1 and to 58 on C_2. To each relayed IRQ multiple LET events can be assigned, leading to table entries with identical relative offsets. Each entry uses 16 byte of memory and can encode multiple events for the same OS task. As an example, the back-2-back activation of two LET tasks consists of a pointer swap for the first and an activation of the second LET task. If those LET tasks are mapped to the same OS task, both events can be encoded with a single table entry. If the OS task differs, maybe because of different periods, two separate entries are needed.

Completely missing in Table 8.6 is the additional memory overhead due to double buffering. Reason for this is the fact, that the needed data was not provided with the example. In order to still be able to evaluate the needed overhead, we will have a look on the use-case from the 2017 WATERS industrial challenge used in [52, 83, 32, 34]. The system consists of 9 periodic tasks, some additional

IRQs and an angle synchronous crankshaft task. For the sake of simplicity we will focus only on the periodic tasks, which can be implemented as a macro-LET system.

Each task produces a set of outputs and takes a set of inputs. According to Section 7.2.3, the output of each task i consists of the written data, stored in a double buffer ($MEM_{OUT,i}$), and a control struct ($MEM_{OUT\_CS,i}$). The input of each task i is defined by a list of used double buffers from other tasks ($MEM_{IN\_List,i}$) and local copies of the corresponding read pointers ($MEM_{IN\_Local,i}$). This leads to

$$\begin{aligned} MEM_{OUT,i} &= 2 \cdot OutputDataBytes \\ MEM_{OUT\_CS,i} &= 2 \cdot sizeof(pointer)Bytes \\ MEM_{IN\_List,i} &= 2 \cdot NumOfInputData \cdot sizeof(pointer)Bytes \\ MEM_{IN\_Local,i} &= NumOfInputData \cdot sizeof(pointer)Bytes \end{aligned}$$

with 4 bytes for the size of a pointer on the AURIX μC. In case of a ringbuffer implementation, which might store more than two values of each global data variable, the term for $MEM_{OUT,i}$ must be modified accordingly.

The periodic tasks from the 2017 WATERS challenge example write overall $14256Bytes$ of output data. If this amount of data is stored in double buffers this leads to:

$$MEM_{OUT} = 2 \cdot 14256Bytes = 28512Bytes \tag{8.4}$$

For each of the 9 tasks a distinct control struct is needed to access the corresponding double buffer, overall this leads to:

$$MEM_{OUT\_CS,} = 9 \cdot 2 \cdot 4Bytes = 72Bytes \tag{8.5}$$

The internal dependencies of the different tasks result in overall 61 different references to double buffers which need to be processed during LET handling and stored as task local pointer copies. This leads to:

$$\begin{aligned} MEM_{IN\_List} &= 2 \cdot 61 \cdot 4Bytes = & 244Bytes \\ MEM_{IN\_Local} &= 61 \cdot 4Bytes = & 488Bytes \end{aligned}$$

The overall memory consumption if then $29316Bytes$ for double buffers and all other control structs. Comparing this to previous data results in a memory overhead of $\sim$ 106%. Judging by the numbers this overhead is dominated by the double buffers and only a fraction of it is due to the needed control structs. Overall the memory overhead is dominated by the application specific data like IRQ/event tables or double buffers.

8.2.2 Overload behavior

As already mentioned, LETs can be interpreted as implicit deadlines. In case of a macro-LET system nothing changes since the deadlines introduced through LET are equal to the previous RMS based system. This is different in case of micro-LET as the LETs are not applied on period boundaries. As a result, micro-LET based systems are more prone to higher priority load. In order to test this for the proposed micro-LET system from Table 8.5 a simple method is constructed to generate high priority interference. C_0 is used to generate periodic core-to-core IRQs to both application cores with different periods. On C_1 and C_2 the corresponding ISR is executed, with a higher priority than the LET controlled OS tasks but a lower than the LET event handling. The executed ISR consumes a defined value of CPU time and returns to the previous execution context. The interference based blocking for each application core C_x is therefore given as:

$$B_{x,max}(\Delta t) = B_{HP,x,min} \cdot \left\lceil \frac{\Delta t}{T_{x,min}} \right\rceil$$

$T_{x,min}$ represents the minimum allowed interarrival time for a higher priority interference of size $B_{HP,x,min}$, under constraint that no LET is missed. $B_{HP,x,min}$ is simply the minimum of all $B_{HP,i}$ on core C_x, which can be calculated according to Figure 6.8 via:

$$B_{HP,i} = \lambda_i - (\overline{C}_i - B_{ACT})$$

Table 8.7 lists all $B_{HP,i}$ terms for both application cores and the three different integration methods (with B_{ACT} rounded up). In case of the LET task setup on C_1, $\lambda_{20,3}$ is the LET task with the smallest available slack, for the BCC and BCC+ implementation this leads to:

$$\begin{aligned} B_{ACT} &= 5,8\mu s \approx 6\mu s \\ B_{HP,1,min} &= B_{HP,20,3} = 250\mu s - (125\mu s + 6\mu s) = 119\mu s \end{aligned}$$

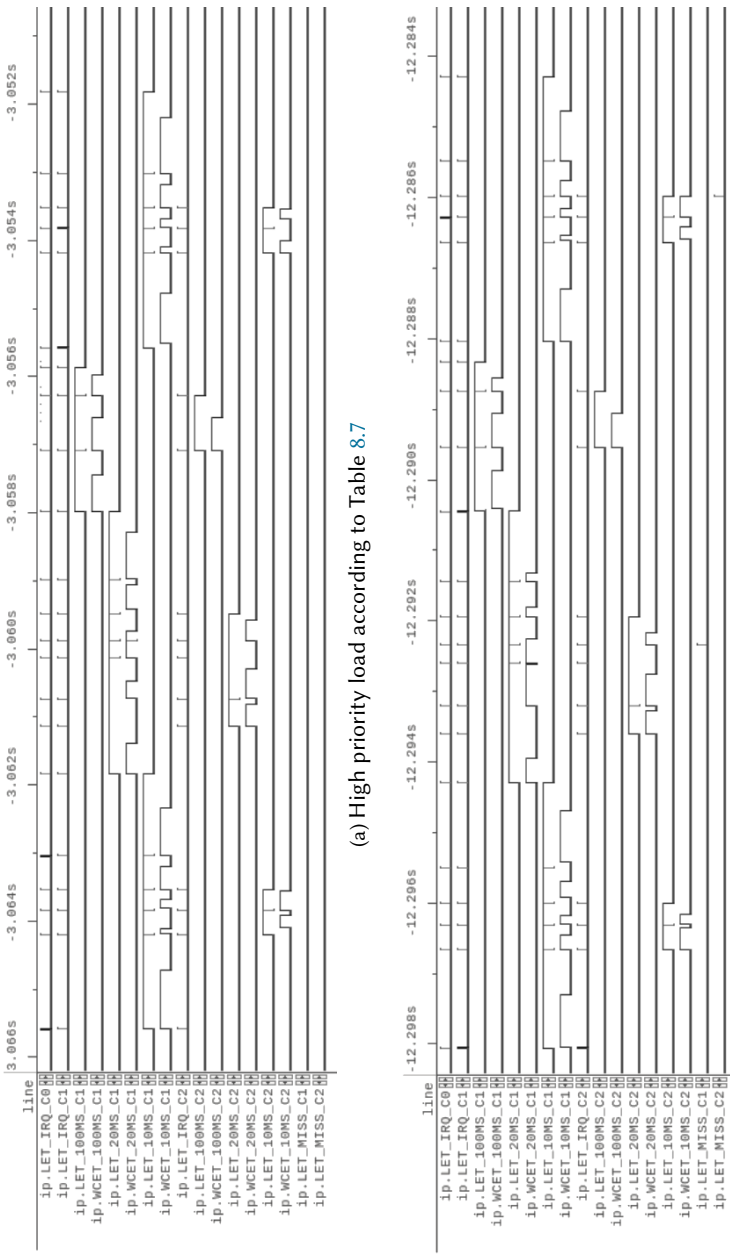
With a given value for $B_{HP,x,min}$, the minimum interarrival time $T_{x,min}$ can be derived. General idea is to find the smallest value for $T_{x,min}$, such that none LET is missed for a periodic interference of $B_{HP,x,min}$ with period $T_{x,min}$ on core C_x. For the given use-case this results in $B_{HP,1,min} = 119\mu s$ every $T_{1,min} = 500\mu s$ on C_1. Performing the same calculations for application C_2 this leads to $B_{HP,2,min} = 140\mu s$ and $T_{2,min} = 400\mu s$. The corresponding $B_{HP,x,min}$ values are highlighted with a bold font in Table 8.7.

Simulating the calculated higher priority load results in Figure 8.9a. The influence of the higher priority load can be seen in the longer times of the *WCET_**

	10 ms					20 ms							100 ms		
λ	10,0	10,1	10,2	10,3	10,4	20,0	20,1	20,2	20,3	20,4	20,5	20,6	100,0	100,1	100,2
LET	1400	350	300	500	1200	700	400	600	250	400	500	1000	900	800	400
Maximum allowed interference on C1															
BCC	766	267	165	273	600	361	-	457	**119**	261	141	531	471	432	209
BCC+	766	267	165	273	600	361	-	457	**119**	261	141	531	471	432	209
ECC	765	266	164	272	599	360	-	456	**118**	260	140	530	470	431	208
Maximum allowed interference on C2															
BCC	-	175	**140**	-	-	-	206	301	-	222	-	-	-	457	-
BCC+	-	175	**140**	-	-	-	206	301	-	222	-	-	-	457	-
ECC	-	174	**139**	-	-	-	205	300	-	221	-	-	-	456	-

Table 8.7: Allowed higher priority interference, times given in [μs]

signals. Those signals represent in this case the response time (since higher priority interference is included) of the corresponding OS task, executed within the LET boundaries. That the calculated higher priority load is at its maximum can be seen for signal *WCET_20MS_C1* at $\sim -3.060s$ where the execution of the corresponding OS task is finished just before the end of the LET visualized by *LET_20MS_C1*. Counting the pulses on *WCET_20MS_C1* shows, that the third executed OS task with period $20ms$ is close to miss the LET. Comparing this with Table 8.7 and 8.5 shows that the corresponding LET task is $\lambda_{20,3}$, which matches the previous interference calculation. Same can be done for the execution on C_2, resulting in $\lambda_{10,2}$. Increasing $B_{HP,x,min}$ further more above the given values from Table 8.7 on both application cores causes the system to miss LETs as shown in Figure 8.9b. The ILM detects an LET miss by checking the OS task state when swapping read and write pointer. If the OS task hasn't finished its execution by that time, a user definable hook function is called. In the shown example the hook function simply toggles an output pin. Future work might include specific mechanisms like dropping activations or suspending the current execution to the next period. At the moment this is an open research topic, since the mechanism to be toggled on an LET miss might highly depend on the actual application. In present work often only an LET miss is signalized but not handled [119]. Also, some ideas like, aborting an OS task during execution and restarting it in the next period, might sound promising but are really hard to implement on existing software and hardware architectures.



(a) High priority load according to Table 8.7

(b) High priority overload with LET misses

Figure 8.9: Overload behavior

"Shut up and take my money!"

- Philip J. Fry

CHAPTER 9
Conclusion

Within this dissertation we provided possible solutions for two known issues in modern automotive ECUs. The primary reason for those issues is the coordination of the newly available computing power in modern automotive μCs or SoCs. Even though the proposed mechanisms could not be more different, both target well-known areas inside a vehicle. Both mechanisms have been developed to target the challenges defined in Section 2.3. On the one hand there is the SPS based budget scheduling, targeting a real-time capable hypervisor which is meant to be part of the AUTOSAR AP. On the other hand, there is the multicore synchronization based on LET, targeting the issues related to the execution of legacy singlecore software. Both mechanisms have been developed and evaluated in order to satisfy the given requirements also given in Section 2.3.

According to Section 2.3.1 a new hypervisor scheduler should be work conserving, with improved response times and provide sufficient temporal isolation with a low profile overhead. In order to provide a novel scheduling mechanism, several mechanisms have been described in Chapter 4 and 5. Comparing the evaluation results from Section 8.1 with the four defined requirements show the extent to which the requirements have been met by the proposed scheduling mechanisms.

With regard to improved response times, all developed scheduling mecha-

nisms improve those in some way. While the IRQ shaping from Section 4.3 only improves the IRQ response times, the SPS-based mechanisms from Chapter 5 also improve task response times as well. Nevertheless, from the formal perspective, only the SPS-based mechanisms with background scheduling are able to improve not only the measured average case but also the formal worst-case response times. This is a side effect of the fact, that only the SPS-based mechanisms with background scheduling provide the possibility of a work-conserving scheduler. While all other proposed mechanisms can not utilize idle times, due to a fix TDMA schedule or budget enforcement, this is possible for a system with background scheduling. In case of sufficient temporal isolation, all mechanisms satisfy Definition 4.2 formally. This is also supported by simulation results and real world measurements in Section 8.1.

The introduced overhead by the mechanisms itself is comparatively low. Nevertheless, the capabilities of the underlying hardware must be taken in account, since all proposed mechanisms might introduce additional context switches. Problematic is at this point not the direct overhead of the context switch, but more the indirect overhead due to caching effects. While the used hardware for evaluation performs the worst due to a virtual indexed cache [28], this is different for architectures of more modern SoCs like the R-Car H3. This is due to mechanisms like an indexing based on physical addresses, address space identifiers or cache pinning.

As the results of Section 8.1 show, the SPS based mechanisms provide all a significantly better performance compared to the standard TDMA scheduling. Even without background scheduling, the SPS based mechanisms provide a much better average case behavior. If the additional computation time during analysis and the additional code overhead for a more sophisticated background scheduling is worth investigating compared to a simple FIFO based queue, heavily depends on the actual application.

Compared to the SPS-based scheduling, the LET implementation targets a different problem. While the SPS-based scheduling only targets a work conserving scheduling on a singlecore, a multicore synchronization is missing. The LET paradigm should be used in order to coordinate automotive software (e.g. control functionality or gateways) on mutlicore platforms. This can be performed on the basis of a standard AUTOSAR CP application without or also on a more sophisticated system with virtualization (e.g. a combination of CP and AP). According to the requirements from Section 2.3.2, the proposed solution should allow a comparable end-to-end latency for effect chains, a minimized input jitter and low profile overhead. If those requirements can be met entirely, depends heavily on the executed application and the way how the LET paradigm is applied to the software.

We described two different mapping solutions discussed by the automotive

community. For better distinction we named those mappings macro-LET and micro-LET. Both mappings minimize the input jitter to zero, simply due to the proposed zero-time communication and the read pointer backup on task activation. Nevertheless, the point in time when the according LET task is started determines the memory overhead needed for double or ring buffers. Also, while micro-LET attempts to provide a comparable or even improved end-to-end latency, this is not the case for macro-LET. Instead, macro-LET only reduces the input jitter to zero.

Nevertheless, this does not mean that micro-LET always provides a perfect solution. Reason for this is the evaluated behavior regarding additional load. In case of macro-LET the deadlines of a task do not change and therefore the system tolerates as much additional load as before, or even more due to distributed execution on different cores. The micro-LET approach is not that flexible due to the shortened implicit deadlines for function blocks. Therefore, the short end-to-end latencies are achieved on the cost of the system's flexibility.

Overall, the static runtime and memory overhead for both proposed mapping solutions is low. The primary generated overhead is for both, runtime and memory, dominated by the application setup and the derived configuration tables as well as memory structures. In general, the LET paradigm allows the coordination of classic automotive software on a multicore platform, and the implementation can be achieved with a manageable overhead as shown in Section 8.2.

Currently missing is an implementation of the proposed LET framework on top of a hypervisor based setup. Even though the current implementation of the ILM does not support virtualization, it should be possible to integrate it with modified hardware and OS ports. From a formal point of view, the analysis modifications from Chapter 6 doesn't need to be modified in order to support an analysis with hierarchical scheduling as described in Chapter 4 and 5. This is possible, since both still rely on a busy window based analysis. Nevertheless, a combined implementation of the proposed SPS based hypervisors scheduling and a synchronization based on LET, can be considered as future work.

APPENDIX A
Publications

This chapter gives an overview about documents and software, published by the author. The publications are subdivided into in reviewed and unreviewed contributions. The entries in each section are sorted chronologically.

A.1 Reviewed

Reference: [28]
Matthias Beckert, Moritz Neukirchner, Rolf Ernst, and Stefan M Petters. Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor. In *Proc. of 51st Annual Design Automation Conference (DAC)*. ACM, 2014

This paper describes the first basic idea regarding the monitoring based IRQ shaping. It highlights the challenges for a fast IRQ handling in a hypervisor based system.

Reference: [23]
Matthias Beckert and Rolf Ernst. Designing time partitions for real-time hypervisor with sufficient temporal independence. In *Proc. of 52st Annual Design Automation Conference (DAC)*. ACM, 2015

As a result of feedback received for [28], this paper describes a way how to design systems with sufficient slack for monitoring based IRQ shaping. Main contribution of this paper is the developed cycle optimization algorithm.

Reference: [27]
Matthias Beckert, Mischa Möstl, and Rolf Ernst. Zero-Time Communication for Automotive Multi-Core Systems under SPP Scheduling. In *Proc. of Emerging Technologies and Factory Automation (ETFA)*, Berlin, Germany, Sep 2016

Based on cooperation with the automotive industry, the paper describes a double buffer based zero-time communication implementation for LET. Additionally, a priority boosting mechanism is proposed in order to allow tighter LETs.

Reference: [26]
Matthias Beckert, Kai Björn Gemlau, and Rolf Ernst. Exploiting Sporadic Servers to provide Budget Scheduling for ARINC653 based Real-Time Virtualization Environments. In *Proc. of Design Automation and Test in Europe (DATE)*, Lausanne, Switzerland, March 2017

While the mechanisms proposed in [28] and [23] still rely on a TDMA based scheduling, this paper proposes an SPS based system as replacement. Primary goal of this paper is, to present a scheduling which provides the same isolation, but a more flexible and less static execution compared to TDMA.

Reference: [24]
Matthias Beckert and Rolf Ernst. Response Time Analysis for Sporadic Server based Budget Scheduling in Real Time Virtualization Environment. *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), ACM Transactions on Embedded Computing Systems ESWEEK Special Issue*, Oktober 2017

While [26] explains the basic architecture of an SPS based system, it neglects the formal RTA. Therefore, this is done in this paper with an additionally proposal for integration and implementation of background scheduling.

Reference: [25]
Matthias Beckert and Rolf Ernst. The IDA LET Machine - An efficient and streamlined open source implementation of the Logical Execution Time Paradigm. In *International Workshop on New Platforms for Future Cars (NPCar)*, March 2018

Like in [27], the cooperation with the automotive industry required the actual implementation of a zero-time communication based LET framework. This workshop contribution provides a short overview of the ILM implementation.

A.2 Unreviewed

Reference: [7]
The IDA-LET-Machine.
https://github.com/matthiasb85/IDA-LET-Machine

As a result of [27] and preceding to [25], the ILM has been developed as an academic testbed. For maximum visibility it was published under GPL v3 with an additional linking exception for closed source software. The referenced link represents the primary repository, which provides the current public version of the ILM.

Reference: [35]
Dagstuhl-Seminar 18092. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems. https://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=18092, 2018

During the referenced Dagstuhl seminar, the ideas proposed in [27] and [25] have been presented and discussed. A short abstract of the contribution is also referenced in the official report [44] of the held seminar.

List of Figures

List of Tables

List of Code

Acronyms

ACE AXI Coherency Extension

ADAS Advanced Driver Assistance Systems

AFDX Avionics Full DupleX Switched Ethernet

AMBA Advanced Microcontroller Bus Architecture

AP Adaptive Platform

APEX Application Executive

API Application Programming Interface

ARINC653 Aeronautical Radio Incorporated Avionics Application Standard Software Interface

ASIL Automotive Safety Integrity Level

AUTOSAR Automotive Open System Architecture

AVB Audio/Video Bridging

AXI Advanced eXtensible Interface

BCC Basic Conformance Class

BCET Best-Case Execution Time

BCRT Best-Case Response Time

BET Bounded Execution Time

BH Bottom Half

BSW Basis Software

CAN Controller Area Network

CAN-FD CAN with Flexible Data-Rate

CB API Callback API

CCU6 Capture Compare Unit 6

COMStack Communication Stack

CPA Compositional Performance Analysis

CP Classic Platform

CPU Central Processing Unit

DC Direct Current

DMA Direct Memory Access

DO-178B DO-178B, Software Considerations in Airborne Systems and Equipment Certification

DO-254 DO-254, Design Assurance Guidance for Airborne Electronic Hardware

DSP Digital Signal Processor

DSPR Data Scratch-Pad SRAM

ECC Extended Conformance Class

ECU Electronic Control Unit

EDF Earliest Deadline First

EEA Electric/Electronic Architecture

FIFO First-In First-out

FIT Faults In Time

FPU Floating Point Unit

GPSRN General Purpose Service Request Node

GTM Generic Timer Module

HAL Hardware Abstraction Layer

HMI Human Machine Interface

IEC61508 International Electrotechnical Commission, Standard 61508: "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems"

ILM IDA LET Machine

IMA Integrated Modular Avionics

IP Internet Protocol

IRQ Interrupt Request

ISA Instruction Set Architecture

ISO26262 International Standard 26262: "Road vehicles - Functional safety"

ISR Interrupt Service Routine

LET Logical Execution Time

LFM Latent Fault Metric

LIN Local Interconnect Network

LMU Local Memory Unit

LPDDR4 SDRAM Low Power Double Data Rate Fourth-Generation Synchronous Dynamic Random-Access Memory

MAC Media Access Controller

MBD Model Based Design

MCAL μC Abstraction Layer

MEMStack Memory Stack

MMU Memory Management Unit

MOST Media Oriented Systems Transport

MPU Memory Protection Unit

OEM Original Equipment Manufacturer

OSEK Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug

OS Operating System

PCB Printed Circuit Board

PDU Protocol Data Unit

PMHF Probabilistic Metric for random Hardware Failures

PMU Program Memory Unit

POSIX Portable OS Interface

POS Partition OS

PSPR Program Scratch-Pad SRAM

PWM Pulse Width Modulated

pyCPA Python Implementation of Compositional Performance Analysis

RAM Random Access Memory

RISC Reduced Instruction Set Computer

RMS Rate Monotonic Scheduling

ROM Read Only Memory

RTA Response Time Analysis

RTC Real-Time Calculus

RTE Runtime Environment

RTOS Real-Time OS

SIL Safety Integrity Level

SoC System On a Chip

SOME/IP Scalable Service-Oriented Middleware over IP

SPFM Single-Point Fault Metric

SPI Serial Peripheral Interface

SPP Static Priority Preemptive

SPS Sporadic Server

SRAM Static Random-Access Memory

SRI System Resource Interconnect

TCB Task Control Block

TCM Tightly Coupled Memory

TDL Timing Definition Language

TDMA Time Division Multiple Access

TH Top Half

μC Microcontroller

VFB Virtual Functional Bus

VM Virtual Machine

VRB Variable Rate-Dependent Behavior

WCET Worst-Case Execution Time

WCRT Worst-Case Response Time

XCP Universal Measurement and Calibration Protocol

ZET Zero Execution Time

Bibliography

[1] ARM. https://www.arm.com.

[2] Automotive open system architecture. https://www.autosar.org/.

[3] Infineon AURIX 32-bit multi core TriCore. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/aurix-safety-joins-performance/.

[4] Infineon AURIX TC27xt. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/aurix-safety-joins-performance/aurix-family-tc27xt/.

[5] Infineon TriCore. https://www.infineon.com/cms/de/product/microcontroller/32-bit-tricore-microcontroller/.

[6] Renesas R-CAR H3. https://www.renesas.com/en-sg/solutions/automotive/products/rcar-h3.html.

[7] The IDA-LET-Machine. https://github.com/matthiasb85/IDA-LET-Machine.

[8] Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proceedings of the 4th ACM*

International Conference on Embedded Software, EMSOFT '04, pages 95–103, New York, NY, USA, 2004. ACM.

[9] ARM. big.LITTLE. https://developer.arm.com/technologies/big-little.

[10] ARM. *ARM Architecture Reference Manual*, 2000.

[11] AUTOSAR Group. AUTOSAR EXP Virtual Functional Bus. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_VFB.pdf, 2017.

[12] AUTOSAR Group. AUTOSAR SWS Communication. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_COM.pdf, 2017.

[13] AUTOSAR Group. AUTOSAR SWS Operating System. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_OS.pdf, 2017.

[14] AUTOSAR Group. AUTOSAR SWS Operating System Adaptive Platform. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-10/AUTOSAR_SWS_OperatingSystemInterface.pdf, 2017.

[15] AUTOSAR Group. AUTOSAR SWS Watchdog Manager. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_WatchdogManager.pdf, 2017.

[16] AUTOSAR Group. Explanation of Adaptive Platform Design. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/17-10/AUTOSAR_EXP_PlatformDesign.pdf, 2017.

[17] Philip Axer. *Performance of Time-Critical Embedded Systems under the Influence of Errors and Error Handling Protocols*. PhD thesis, TUBS, 2015.

[18] Philip Axer, Moritz Neukirchner, Sophie Quinton, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-time analysis of parallel fork-join workloads with real-time constraints. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[19] S. Balakrishnan. The linux operating system. *Resonance*, 4(4):64–72, Apr 1999.

[20] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2010.

[21] Markus Bechter. AUTOSAR Adaptive Platform Key concepts and development process. https://www.autosar.org/fileadmin/AOC/AOC_2016/Presentations/AUTOSAR_9AOC_Adaptive_Platform_BECHTER.pdf, 2016.

[22] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 80:104–113, 2017.

[23] Matthias Beckert and Rolf Ernst. Designing time partitions for real-time hypervisor with sufficient temporal independence. In *Proc. of 52st Annual Design Automation Conference (DAC)*. ACM, 2015.

[24] Matthias Beckert and Rolf Ernst. Response Time Analysis for Sporadic Server based Budget Scheduling in Real Time Virtualization Environment. *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), ACM Transactions on Embedded Computing Systems ESWEEK Special Issue*, Oktober 2017.

[25] Matthias Beckert and Rolf Ernst. The IDA LET Machine - An efficient and streamlined open source implementation of the Logical Execution Time Paradigm. In *International Workshop on New Platforms for Future Cars (NPCar)*, March 2018.

[26] Matthias Beckert, Kai Björn Gemlau, and Rolf Ernst. Exploiting Sporadic Servers to provide Budget Scheduling for ARINC653 based Real-Time Virtualization Environments. In *Proc. of Design Automation and Test in Europe (DATE)*, Lausanne, Switzerland, March 2017.

[27] Matthias Beckert, Mischa Möstl, and Rolf Ernst. Zero-Time Communication for Automotive Multi-Core Systems under SPP Scheduling. In *Proc. of Emerging Technologies and Factory Automation (ETFA)*, Berlin, Germany, Sep 2016.

[28] Matthias Beckert, Moritz Neukirchner, Rolf Ernst, and Stefan M Petters. Sufficient Temporal Independence and Improved Interrupt Latencies in a Real-Time Hypervisor. In *Proc. of 51st Annual Design Automation Conference (DAC)*. ACM, 2014.

[29] Chris Berg and Frank Walkenbach. Uniform Hardware Platforms for all Car Features. https://www.sysgo.com/fileadmin/user_upload/www.sysgo.com/fachartikel/Uniform_hardware_platforms_for_all_car_features.pdf, 2016.

[30] Pierre-Antoine Bernard and Cornel Izbasa. Virtualization Solutions for the AUTOSAR Classic and Adaptive Platforms. https://www.autosar.org/fileadmin/AOC/AOC_2017/Presentations/AOC2017_Bernard_Izbasa.pdf, 2017.

[31] A. Biondi and M. Di Natale. Achieving predictable multicore execution of automotive applications using the let paradigm. In *2018 IEEE Real-Time*

and Embedded Technology and Applications Symposium (RTAS), pages 240–250, April 2018.

[32] Alessandro Biondi, Paolo Pazzaglia, Alessio Balsini, and Marco Di Natale. Logical execution time implementation and memory optimization issues in autosar applications for multicores. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Dubrovnik, Croatia, jun 2017.

[33] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *Proc. of 7th European conference on Computer Systems*, pages 323–336. ACM, 2012.

[34] Frederic Boniol, Julien Forget, and Claire Pagetti. Waters industrial challenge 2017 with prelude. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Dubrovnik, Croatia, jun 2017.

[35] Dagstuhl-Seminar 18092. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems. https://www.dagstuhl.de/de/programm/kalender/semhp/?semnr=18092, 2018.

[36] Matthew Danish, Ye Li, and Richard West. Virtual-cpu scheduling in the quest operating system. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011.

[37] Robert I Davis and Alan Burns. Hierarchical fixed priority pre-emptive scheduling. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10–pp. IEEE, 2005.

[38] Robert I Davis, Timo Feld, Victor Pollex, and Frank Slomka. Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 51–62. IEEE, 2014.

[39] K Devika and R Syama. An overview of autosar multicore operating system implementation. *International Journal of Innovative Research in Science, Engineering and Technology*, 2:3162–3169, 2013.

[40] Jonas Diemer, Philip Axer, and Rolf Ernst. Compositional performance analysis in python with PyCPA. *Proc. of WATERS*, 2012.

[41] Embedded Office. μC/OS-MMU. http://www.embedded-office.com/en/products/uC-OS-MMU.html.

[42] Paul Emberson, Roger Stafford, and Robert I Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[43] Tom Erkkinen and Mirko Conrad. Safety-critical software development using automatic production code generation. Technical report, SAE Technical Paper, 2007.

[44] Rolf Ernst, Stefan Kuntz, Sophie Quinton, and Martin Simons. The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092). *Dagstuhl Reports*, 8(2):122–149, 2018.

[45] Dario Faggioli, Antonio Mancina, Fabio Checconi, and Giuseppe Lipari. Design and implementation of a posix compliant sporadic server for the linux kernel. In *Proceedings of the 10th Real-Time Linux workshop*, 2008.

[46] Emilia Farcas, Claudiu Farcas, Wolfgang Pree, and Josef Templ. Transparent distribution of real-time components based on logical execution time. *ACM SIGPLAN Notices*, 40(7):31–39, 2005.

[47] G Farrall, C Stellwag, J Diemer, and R Ernst. Hardware and software support for mixed-criticality multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe, WICERT, DATE*, 2013.

[48] Goran Frehse, Arne Hamann, Sophie Quinton, and Matthias Woehrle. Formal analysis of timing effects on closed-loop properties of control software. In *Real-Time Systems Symposium (RTSS), 2014 IEEE*, pages 53–62. IEEE, 2014.

[49] Jon Friedman. Matlab/simulink for automotive systems design. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 87–88. European Design and Automation Association, 2006.

[50] Simon Fürst and Markus Bechter. AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform. In *Dependable Systems and Networks Workshop, 2016 46th Annual IEEE/IFIP International Conference on*, pages 215–217. IEEE, 2016.

[51] Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, and Luca Abeni. Architecture for a portable open source real time kernel environment. In *In Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.

[52] Kai-Björn Gemlau, Johannes Schlatow, Mischa Möstl, and Rolf Ernst. Compositional analysis for the waters industrial challenge 2017. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Dubrovnik, Croatia, jun 2017.

[53] David B Golub, Daniel P Julin, Richard F Rashid, Richard P Draves, Randall W Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and mach. In *In Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 11–30, 1992.

[54] Arne Hamann, Dakshina Dasari, Simon Kramer, Michael Pressler, Falk Wurst, and Dirk Ziegenbein. Waters industrial challenge 2017. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Dubrovnik, Croatia, jun 2017.

[55] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[56] BR Heap. Permutations by interchanges. *The Computer Journal*, 6(3):293–298, 1963.

[57] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proc. of 1st asia-pacific workshop on Workshop on systems*. ACM, 2010.

[58] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis–the SymTA/S Approach. *IEEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[59] Julien Hennig, Hermann von Hasseln, Hassan Mohammad, Stefan Resmerita, Stefan Lukesch, and Andreas Naderlinger. Towards parallelizing legacy embedded control software using the LET programming paradigm. In *Proc. of WiP Papers of the 22nd IEEE Real-Time and Embedded Technology and Apllications Symposium (RTAS2016)*, 2016.

[60] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree. From control models to real-time code using giotto. *IEEE Control Systems Magazine*, 23(1):50–64, Feb 2003.

[61] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Embedded Software*, pages 166–184, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[62] Thomas A Henzinger and Christoph M Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):33, 2007.

[63] Thomas A Henzinger, Christoph M Kirsch, and Slobodan Matic. Schedule-carrying code. In *Embedded Software*, pages 241–256. Springer, 2003.

[64] Infineon. Automotive application guide.

[65] International Electrotechnical Commission. IEC61508 Ed.2 - Functional Safety of Electrical/Electronic/Programmable Safety-related Systems, 2008.

[66] International Standardization Organization. ISO26262 - Road Vehicles. Functional Safety, 2011.

[67] Christian Jann. Koexistenz von AUTOSAR Softwarekomponenten und Linux-Programmen für zukünftige High Performance Automotive Steuergeräte. Master's thesis, TU Cemnitz, 2016.

[68] Jean J. Labrosse. *MircoC/OS-II The Real Time Kernel*. CMP, 2002.

[69] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[70] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS Microkernel. In *Proc. of 1st International Workshop on Microkernels for Embedded Systems (MIKES)*, 2007.

[71] Brian W Kernighan, Dennis M Ritchie, Clovis L Tondo, and Scott E Gimpel. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[72] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for I/O performance. In *Proc. of 5th SIGPLAN/SIGOPS international conference on Virtual execution environment*, pages 101–110. ACM, 2009.

[73] Christoph M. Kirsch. Principles of real-time programming. In Alberto Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software*, pages 61–75, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[74] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

[75] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[76] Y-H Lee, Daeyoung Kim, Mohamed Younis, and Jeff Zhou. Partition scheduling in apex runtime environment for embedded avionics software. In *Proc. of 5th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 1998.

[77] JP Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, 1990.

[78] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of 11th Real-Time Systems Symposium (RTSS)*, pages 201–209. IEEE, 1990.

[79] Ye Li, Matthew Danish, and Richard West. Quest-V: A virtualized multikernel for high-confidence systems. *arXiv preprint arXiv:1112.5136*, 2011.

[80] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[81] Luxoft. Automotive timing analysis solution. https://auto.luxoft.com/uth/timing-analysis-tools/, 2018.

[82] J. Martinez, I. Sañudo, and M. Bertogna. Analytical characterization of end-to-end communication delays with logical execution time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2018.

[83] Jorge Martinez, Ignacio Sanudo, Paolo Burgio, and Marko Bertogna. End-to-end latency characterization of implicit and let communication models. In *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, Dubrovnik, Croatia, jun 2017.

[84] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. https://securityzap.com/files/RemoteCarHacking.pdf, 2015.

[85] MISRA Consortium and MISRA, C and others. Guidelines for the use of the C language in critical systems, 2013, 2012.

[86] Aurélien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. Multisource software on multicore automotive ECUs combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, 2012.

[87] Moritz Neukirchner. *Establishing Sufficient Temporal Independence Efficiently - A Monitoring Approach*. PhD thesis, Technische Universität Braunschweig, Braunschweig, Germany, Jul 2014.

[88] Moritz Neukirchner. Building Performance ECUs with Adaptive AUTOSAR. https://www.autosar.org/fileadmin/AOC/AOC_2017/Presentations/AOC2017_Neukirchner.pdf, 2017.

[89] Moritz Neukirchner, Tobias Michaels, Philip Axer, Sophie Quinton, and Rolf Ernst. Monitoring arbitrary activation patterns in real-time systems. In *Proc. of 33rd Real-Time Systems Symposium (RTSS)*. IEEE, 2012.

[90] Silviu-Iulian Niculescu. *Delay effects on stability: a robust control approach*, volume 269. Springer Science & Business Media, 2001.

[91] Diego Ongaro, Alan L Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proc. of 4th SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 1–10. ACM, 2008.

[92] OSEK Group. OSEK/VDX Operating System Specification 2.2.3, 2015.

[93] P.J. Prisaznuk. ARINC 653 role in Integrated Modular Avionics (IMA). In *Proc. of Digital Avionics Systems Conference (DASC)*, 2008.

[94] Radio Technical Commission for Aeronautics. DO-178B - software considerations in airborne systems and equipment certification, Dec 1992.

[95] Radio Technical Commission for Aeronautics. DO-254 - design assurance guidance for airborne electronic hardware, Apr 2000.

[96] John Regehr and Usit Duongsaa. Preventing interrupt overload. In *ACM SIGPLAN Notices*, volume 40, pages 50–58. ACM, 2005.

[97] Renesas. R-Car H3/M3 Device manual.

[98] S. Resmerita, A. Naderlinger, M. Huber, K. Butts, and W. Pree. Applying real-time programming to legacy embedded control software. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 1–8, April 2015.

[99] Kai Richter. *Compositional Scheduling Analysis Using Standard Event Models*. PhD thesis, Technische Universität Braunschweig, 2004.

[100] Roger Stafford. Random Vectors with Fixed Sum . http://www.mathworks.com/matlabcentral/fileexchange/9700, 2018.

[101] Rutronik. Microcontroller Technologies. https://www.rutronik.com/fileadmin/Rutronik/Downloads/printmedia/products/01_semiconductors/microcontroller_technologies_en.pdf.

[102] Saowanee Saewong, Ragunathan Rajkumar, John P Lehoczky, and Mark H Klein. Analysis of hierar hical fixed-priority scheduling. In *ECRTS*, volume 2, page 173, 2002.

[103] Alberto Sangiovanni-Vincentelli and Marco Di Natale. Embedded system design for automotive applications. *Computer*, 40(10), 2007.

[104] Simon Schliecker, Jonas Rox, Matthias Ivers, and Rolf Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *Proc. of 6th International Conference on Hardware Software Codesign and System Synthesis(CODES+ISSS)*, 2008.

[105] Robert Sedgewick. Permutation generation methods. *ACM Computing Surveys (CSUR)*, 9(2):137–164, 1977.

[106] Tobias Sehnke, Matthias Schultalbers, and Rolf Ernst. Temporal properties in component-based cyber-physical systems. In *Proceedings of the 9th Embedded Real Time Software and Systems*, 2018.

[107] Aeronautical Radio Inc Software. ARINC Specification 653-1. Avionics application standard interface, 2018.

[108] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1(1), 1989.

[109] Mark Stanovich, Theodore P Baker, An-I Wang, and Michael González Harbour. Defects of the POSIX sporadic server and how to correct them. In

Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE. IEEE, 2010.

[110] Steffen Stein. *Allowing Flexibility in Critical Systems: The EPOC Framework*. PhD thesis, TU BS, 2012.

[111] Domitian Tamas-Selicean and Paul Pop. Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. In *Proc. of 14th International Symposium on Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 1–10. IEEE, 2011.

[112] Josef Templ. TDL specification and report. Technical report, Department of Computer Science, University of Salzburg, 2004.

[113] The IEEE and The Open Group. Base Specifications Issue 7, IEEE Std 1003. http://pubs.opengroup.org/onlinepubs/9699919799/, 2017.

[114] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Circuits and Systems, 2000. Proceedings. ISCAS 2000 Geneva. The 2000 IEEE International Symposium on*, volume 4, pages 101–104 vol.4, 2000.

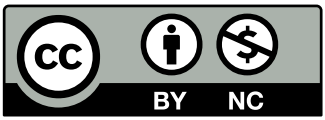
[115] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. Explaining packet delays under virtualization. *ACM SIGCOMM Computer Communication Review*, 41(1):38–44, 2011.

[116] WIND RIVER. WIND RIVER VxWORKS 653 Platform. https://www.windriver.com/products/product-overviews/vxworks-653-product-overview/vxworks-653-product-overview.pdf, 2018.

[117] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: towards real-time hypervisor scheduling in xen. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*. IEEE, 2011.

[118] Yuchen Zhou, Thomas E Fuhrman, and Prathap Venugopal. Scheduling Techniques for Automated Driving Systems using the AUTOSAR Adaptive Platform. https://www.autosar.org/fileadmin/AOC/AOC_2017/Presentations/AOC2017_Venugopal.pdf, 2017.

[119] Dirk Ziegenbein and Arne Hamann. Timing-aware control software design for automotive systems. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.



https://doi.org/10.24355/dbbs.084-201911070747-5